Université catholique de Louvain
School of Applied Sciences
Department of Computer Science

Elaborating Complete and Consistent Requirements for Security-Critical Systems

Renaud De Landtsheer

Thèse présentée en vue de
l'obtention du grade de
Docteur en Sciences Appliquées

Members of Jury:

Prof. **Marc Dacier** (Institut Eurécom, France)
Prof. **Yves Deville** (UCL-Departement of Computer Science), President
Prof. **Baudouin Le Charlier** (UCL-Departement of Computer Science)
Prof. **Bashar Nuseibeh** (Open University, UK)
Prof. **Charles Pecheur** (UCL-Departement of Computer Science)
Prof. **Axel van Lamsweerde** (UCL-Departement of Computer Science), Supervisor

June 2007

Abstract

The elaboration of requirements is a crucial step in the development of software-intensive security-critical application. From this stage on, the security engineering process heads towards the implementation of the identified security requirements. A missing, inadequate, imprecise or inconsistent requirement might cause the resulting application to have vulnerabilities, despite the huge amount of work that might be invested in implementing the elaborated security requirements.

Vulnerabilities in requirements models notably arise from incompleteness and inconsistency. Incompleteness refers to missing requirements; inconsistency refers to requirements prescribing incompatible behaviors. Inconsistent requirements might give rise to vulnerabilities because they cannot be enforced altogether: one of them will be violated at run time.

This thesis proposes three techniques for assisting engineers in the elaboration of complete and consistent requirements through model-based analysis. The two first focus on completeness issues, the last one on consistency. The three techniques build on a specific model-based framework for goal-oriented requirements analysis. They are demonstrated through models of an e-purse system and an e-signature system.

The first technique is a formal framework for specifying and analyzing confidentiality concerns. Our framework makes it possible to discover confidentiality violations at requirements engineering time through model checking.

The second technique focuses on the responsibility assignment of security requirements. Some agents in the software or in its environment might be malicious or too vulnerable. One should not rely on them in the enforcement of security goals. This technique introduces a security-specific completeness criterion ensuring that the enforcement of security goals only depends on the behavior of agents that are known to be trustworthy or built to be trustworthy. A systematic process is proposed for elaborating requirements models complying with this criterion.

The third technique makes it possible to automatically detect goal-level inconsistencies called divergences. Divergences between goals lead to inconsistencies between requirements, and to potential vulnerabilities in the running system.

Acknowledgments

I am grateful to Axel van Lamsweerde, my supervisor, for useful comments on earlier versions of this thesis, and for his support during this research. I am also grateful to Charles Pecheur, Bashar Nuseibeh, Marc Dacier, and Baudouin Le Charlier for their useful comments on earlier versions of this thesis. I also thank Peter van Roy for attending my private defense.

I want to especially thank Charles Pecheur and Yves Deville for their encouragements, notably at the very end of my thesis, when the end seemed to be further away than ever.

I am grateful to Emmanuel Letier, Christophe Damas, Bernard Lambeau, and David Janssens for useful discussions about goal-oriented requirements engineering and security. I also mention Simon Brohez with whom I had some interesting discussion at the very beginning of this thesis.

I am also grateful to CEDITI team, and particularly to Jean-Luc Roussel and Cédric Neve who developed the Objectiver tool that I intensively used for writing the requirements model appearing in this thesis. I must also mention Emanuelle Delor, who told them about beta testing. Thank to her beta-testing process, I was granted a licence of the Objectiver tool.

I am grateful to the CETIC team and particularly to Andre Rifaut for interesting suggestions about half-baked ideas all along the two first years of my PhD. I must also mention Christophe Ponsart, Philippe Massonet, and Jean-François Molderez.

I am also grateful to Raphaël Collet, Isabelle Dony and Grégoire Dooms for productive discussions about constraint programming, and formal verification.

I also want to thank Pierre Schaus and Jean-Noël Monette for giving me an extra desk in their office.

I am also grateful to the geeks, nerds, and colleagues of the Reaumur building, and particularly of the third floor, for their encouragements. I recall their name here: Sébastien Barré, Aurélien Bonnet, Johan Brichau, Jérôme Callut, Raphaël Collet, Christophe Damas, Cédric de Launois,

Benoît Donnet, Isabelle Dony, Grégoire Dooms, Pierre François, François Gobert, Sebastián González, Thomas Goor, David Janssens, Yves Jaradin, Thibaut Helleputte, Dung Kieu Huu, Bernard Lambeau, Damien Leroy, Emmanuel Letier, Xavier Martin, Boris Mejias, Jean-Noël Monette, Diego Ordonez, Gustavo Ospina, Cristel Pelsser, Dung Pham Quang, Luis Quesada, Bruno Quoitin, Pierre Schaus, Fred Spiessens, Sébastien Tandel, Steve Uhlig, Virginie Van den Schrieck, Nicolas Vanderavero, José Vander Meulen, Sébastien Vast, Vu Minh Chieu, and Stephane Zampelli.

I am grateful to the academic staff of our department: Olivier Bonaventure, Pierre-Jacques Courtois, Philippe Delsarte, Yves Deville, Pierre Dupont, Baudouin Le Charlier, Marc Lobelle, Kim Mens, Elie Milgrom, Charles Pecheur, Michel Sintzoff, André Thayse, Axel van Lamsweerde, and Peter Van Roy.

I am grateful to the administrative staff of our department: Claude Daubies, Stéphane Deconinck, Viviane Dehut, Fabienne Delbrouck, Phan Man Tien, Christine Jacqmot, Chantal Poncin, Pierre Reinbold, and Stéphanie Remacle-Landrain. I especially want to thank Stéphanie Remacle-Landrain and Viviane Dehut for being very proactive secretaresses and understanding persons and Fabienne Delbrouck for her permanent happiness and shoving sense of humor.

I am grateful to Valérie, my fiancée who encouraged me all along the years that led to this thesis.

I am grateful to my family, to my parents Luc and Marie-Paule, to my brothers Quentin and Gauthier, to my grandparents, uncles, aunts, godfather, godmother, and to the family of my fiancée. I am also grateful to my friends.

The work reported herein was supported by the belgian "Fonds National de la Recherche Scientifique" (FNRS) and b y the Regional Government of Wallonia (MILOS project, RW Conv. 114856).

Contents

Abstract **1**

Acknowledgments **2**

Contents **4**

1 Introduction **8**
1.1 Context and Motivation 8
1.2 Objectives of the Thesis 11
1.3 Overview of Contributions 12
1.3.1 Reasoning about confidentiality at requirements time . 12
1.3.2 Assigning responsibilities for security requirements . . 13
1.3.3 Detecting divergences in goal models 14
1.4 Organization of the Thesis 15

2 Engineering Security-Critical Systems **16**
2.1 Two Examples of Security-Critical Applications 16
2.1.1 An Electronic Payment System 16
2.1.2 An Electronic Signature System 16
2.2 Different Flavours of Security 17
2.3 Structuring Security-Critical Software Applications 17
2.3.1 Cryptographic Layer 17
2.3.2 Protocol Layer 18
2.3.3 Language and System Layer 19
2.3.4 Application Layer 20
2.4 Discussion 30

3 Goal-Oriented Requirements Engineering **33**
3.1 Why the KAOS Goal-Oriented Method? 33
3.2 The KAOS Requirement Specification Language 34
3.2.1 Conceptual Modeling 34
3.2.2 Structuring the Framework 36
3.2.3 Graphical Representation 37
3.2.4 Object Model 37

3.2.5 Goal Model 40
3.2.6 Agent Responsibility Model 45
3.2.7 Agent Interface Model 46
3.2.8 Consistency Rules 47
3.2.9 Overview of the full framework 50
3.3 Goal-Oriented Requirements Elaboration Method 50
3.3.1 Identifying Preliminary Goals 50
3.3.2 Identifying Objects from Preliminary Goals 51
3.3.3 Eliciting new Goals through How and Why Questions 52
3.3.4 Detecting and Correcting Conflicts 53
3.3.5 Assigning Responsibilities 54
3.3.6 Anticipating Abnormal Behaviors: Obstacle Analysis . 55
3.4 Summary 57

4 Reasoning about Confidentiality 58
4.1 Motivation: Analyzing Confidentiality Requirements 58
4.2 Background: Logics For Reasoning About Knowledge 60
4.3 What Do Unauthorized Agents Know? 62
4.3.1 Towards a most powerful attacker model 62
4.3.2 Shopping for axioms and security principles 63
4.4 Specification Patterns For Confidentiality Requirements . . . 65
4.4.1 Degree of approximate knowledge 66
4.4.2 Timing of confidentiality 67
4.5 Checking Confidentiality Claims on Requirements Models . . 69
4.5.1 CONCHITA as seen from outside 69
4.5.2 Inside CONCHITA 74
4.5.3 Properties of CONCHITA 96
4.6 Managing Confidentiality Requirements With CONCHITA . 97
4.6.1 A Confidentiality Enforcement Process and Tactics . . 97
4.6.2 Illustrating the Process and Tactics 98
4.6.3 Confidentiality and Declarativeness 103
4.7 Related Work 104
4.8 Conclusion 104
4.9 Open Issues 105

5 Assigning Responsibilities for Security Requirements 107
5.1 Overview 107
5.2 Trustworthiness in Requirements Models 110
5.2.1 Notation 110
5.2.2 A Goal-oriented Model of Trustworthiness 110
5.2.3 Representing Trustworthiness Graphically 112
5.2.4 Completeness of Security Requirements 112
5.3 Handling Untrustworthy Assignments 113
5.3.1 Capturing Untrustworthiness of Assignments 113

5.3.2 Countermeasures to Untrustworthy Assignments . . . 114
5.3.3 From local security to global security 115
5.3.4 Classifying countermeasures 117
5.4 Ensuring Requirements Completeness 120
5.4.1 Process Overview 120
5.4.2 Detailing the Steps of the Process 120
5.5 Automated Detection of Untrustworthy Assignments 124
5.5.1 Motivation 124
5.5.2 Process Overview 125
5.5.3 Detecting Untrustworthiness with the Alloy Tool . . . 126
5.5.4 Discussion 132
5.6 Evaluation 133
5.6.1 Methodology of the Comparison 133
5.6.2 Result 134
5.6.3 Analysis 134
5.6.4 Discussion 138
5.7 Related Work 138
5.8 Conclusion 139
5.9 Open Issues 140

6 Detecting Divergences in Goal Models 142
6.1 Inconsistencies as Vulnerabilities 142
6.2 Introducing a Formally Simpler Example 144
6.3 Problems with the Formal Definition of Divergence 146
6.4 Refining the Formal Definition of Divergence 148
6.4.1 Towards a More Precise Definition of Divergence . . . 148
6.4.2 Approximating Controlled Variables at Goal Level . . 149
6.4.3 A Refined Definition for Divergence 151
6.4.4 Validating the Refined Definition of Divergences . . . 151
6.4.5 Root and Implied Divergences 153
6.4.6 Organizing Boundary Conditions 155
6.5 Detecting Divergences 156
6.5.1 Problem Statement 156
6.5.2 Background on Divergences Detection Techniques . . 156
6.5.3 Structuring a Technique for Detecting Divergences . . 157
6.5.4 Testing the Realizability of Temporal Assertions . . . 160
6.5.5 Choosing Candidate Sets of Goals 168
6.6 Illustration on Prototype Implementation 172
6.6.1 SSV : a Prototype for Detecting Divergences 173
6.6.2 Implementation Details of SSV 174
6.6.3 Analysing a train control system with SSV 176
6.6.4 Efficiency measurements 188
6.7 Related Work 190
6.8 Conclusion 191

6.9 Open Issues 192

7 Conclusion **195**
7.1 What Makes Security-Critical Systems so Different? 195
7.2 Linking Modeling Technique to Expert Knowledge 196
7.3 Open Issues 197
7.3.1 Making Formal Specification More Profitable 197
7.3.2 Deriving Requirements for Security Protocols 198
7.4 Summary of Contributions 199
7.4.1 Reasoning about confidentiality at requirements time . 199
7.4.2 Assigning responsibilities for security requirements . . 199
7.4.3 Detecting divergences in goal models 200

Bibliography **215**

List of Figures **216**

Index **218**

A Annex of Chapter 4 **221**
A.1 BNF Grammar of CONCHITA 221
A.1.1 Object Model and Base Types 222
A.1.2 Instantiation directives 223
A.1.3 Goals 224
A.1.4 Assertions 224
A.2 Proving the Soundness of the Temporal Layer of CONCHITA 230
A.2.1 Semantics of Temporal Assertions 231
A.2.2 Underlying assumptions 233
A.2.3 Starting the Proof 235
A.2.4 Bounding Time with Strengtening 236
A.2.5 Bounding Time With Weakening 255
A.3 Bounding Integers in CONCHITA 264
A.3.1 Defining the Abstraction Functions 266
A.3.2 Abstracting Expressions 266
A.3.3 Abstracting Assertions on Integers with Strengtening . 277
A.3.4 Abstracting Assertions on Integers with Weakening . . 284
A.4 Input Files for the e-Signature System 291

B Annex of Chapter 5 **301**
B.1 Soundness of the Local Axioms 301

Chapter 1

Introduction

This chapter introduces the context, motivations, objectives and contributions of this thesis.

Section 1.1 presents the context and motivations of this thesis; Section 1.2 presents the objectives of this thesis; Section 1.3 introduces the contributions of this thesis; Section 1.4 presents the structure of this dissertation.

1.1 Context and Motivation

We increasingly need software in industry and business applications. However, software products generally lack quality; we thus increasingly rely upon unreliable software. We can expect to face a major crisis in the future, especially due to security problems. Security problems arise from violations of confidentiality, privacy, integrity and availability of information in software applications. Such violations can be the consequence of external and generally malevolent interventions on unreliable software. They can also be the consequence of incidental events, such as agent malfunction or human errors.

According to CERT, the main Internet security expertise organization, the number of security incidents has been growing exponentially over the last ten years. The number of such incidents has grown from 252 in 1990 to 21,756 in year 2000, and 137,529 in 2003. In 2003, CERT decided to change its statistics because attacks are increasingly carried on using automated tools [CER]. Because of the use of automated tools, a single vulnerability can lead to numerous security incidents. Newer statistics reports on the number of reported vulnerabilities. In 2006 the number of reported vulnerabilities reached 8,064, that is around 31 vulnerabilities reported per working day, not counting the unreported ones. The statistics of reported vulnerabilities are

presented in Figure 1.1. Some attacks have led to medical record corruption leading to death of patients. There is a critical need for development and analysis techniques that will enable us to develop reliable, secured software products.

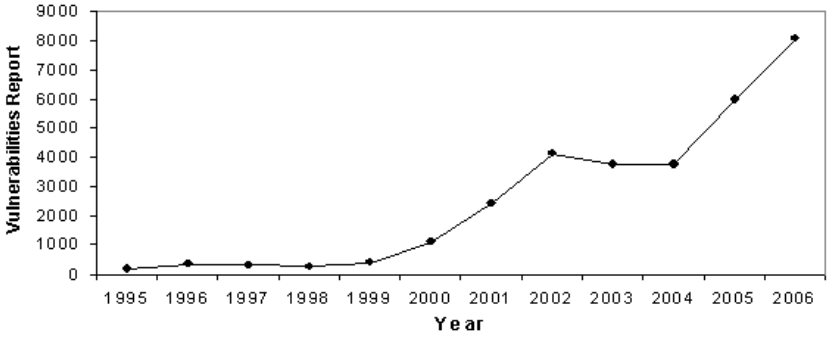

Figure 1.1: Statistics of Reported Vulnerabilities [CER]

Security concerns range over a number of distinct architectural layers represented in Figure 1.2 [Win98]. The first layer is the cryptographic layer that provides basic cryptographic primitives such as crypt, decrypt, hash and nonce generation. The second layer is the protocol layer, grouping standard TCP-like protocols as well as cryptographic protocols. The third layer is the language and system layer. It provides standard services such as secure file exchange. This layer is implemented in programming languages such as C or Java. The top layer is the application layer; it provides the specific application services such as electronic payment. This layer should use the lower layer services to enforce its application-specific security requirements.

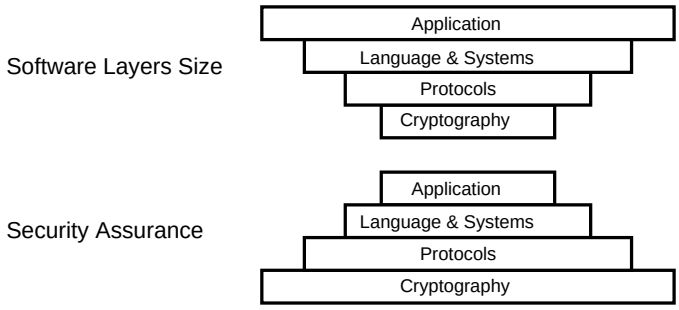

Figure 1.2: Software layers size compared to their security assurance [Win98]

There is some general agreement that security assurance for a layer is inversely proportional to the size of the layer (see Figure 1.2) [Win03, Win98, Sch97, And01]. Cryptography is a mature domain, with plenty of tech-

niques to precisely define which properties are guaranteed and which ones are not. At the protocol level, there are some dedicated logics [BAN89] that allow us to reason about protocols and automated tools supporting these logics [KMM94, Wor97, oSET00]. The latter enables us to verify protocol properties against potential attackers and sometimes to break protocols [Low96, CJM00]. At the system level, there are well-established secure services such as SSH, SSL and HTTPS that offer secure high-level communications facilities with authentication and encryption. There are also technologies such as Authenticode, Active X and Java that intend to offer some security levels, but are the target of many attacks such as denial of service or deception. At the application level, there are virtually no techniques or methods to ensure that specific security requirements on a particular application have been thoroughly identified and are guaranteed by the security policy [Win03, Kem03, Win98, Sch97, And01, VM01, And93, WG05].

This thesis addresses security concerns at the application level. At this level, services such as, for example, web-based banking operations must implement application-specific security requirements in terms of primitives from the lower levels. Typical application-specific security breaches would allow malicious users of a web banking application to know account numbers together with their associated PIN code, or malicious users of an e-commerce application to get some item without having paid for it.

The application layer defines the required behavior of the agents of the system and, more specifically, of the software agents. Agents are active components forming the system. They can be humans, devices, legacy software, etc. This behavior is defined by a set of requirements including functional requirements and security requirements. *Requirements* are prescriptive statement of intent about the system-to-be. *Functional requirements* define the behavior of the system-to-be. *Security requirements* contribute to protecting assets against non-authorized use [HLMN06, vL04]. Our focus is on security requirements here.

Security requirements are generally organized into the following non-exhaustive taxonomy known as the *C.I.A.* taxonomy [CC99, Amo94, Kem03]:

- **Confidentiality** requirements prescribe that sensitive information be not disclosed to unauthorized agents [Kem03] . The term *authorized agent* denotes an agent exposed to information disclosure. *Disclosure* refers to undesired knowledge states of such agents. It thus imposes restrictions on what agents may or may not know about the application over time.
- **Integrity** requirements prescribe that information is changed only in a specified and authorized manner.
- **Availability** requirements prescribe that the system must be accessi-

ble and usable upon demand by some authorized entity.

For security assurance, a set of security requirements should be elaborated, made fully precise, and analyzed against consistency and completeness at requirements engineering time. Completeness ensures that there are "enough" security requirements to ensure that the system is "properly secured" (in a sense to be made precise). Consistency ensures that the described requirements model can be implemented in the sense that there is no requirement prescribing incompatible behaviors to be implemented by a single agent. Notably, security requirements tend to be inconsistent with functional requirements. A typical way of resolving such inconsistencies is to weaken the functional requirements accordingly.

It is not yet very clear how to identify security requirements systematically through the various stages of the requirements engineering process [vL04]. Security requirements are sometimes left unspecified, notably because of the cost of identifying them and because they are not clearly visible in the implementation, unlike functional requirements [CILN02]. The oldest practices rely on general guidelines, partial reuse of existing requirements model, or peer reviewing. Two other trends have emerged, the first one relies on attack identification and prevention and the second one relies on refinement of the desired security properties. The latter might be implicitly captured into threats on the system, which are negation of these desired properties.

1.2 Objectives of the Thesis

Our overall objective is to propose techniques for elaborating requirements models such that the resulting applications do not have vulnerabilities originating from it. The overall direction of the thesis is to build models of the requirements and to reason about them.

Vulnerabilities in requirements models notably arise because of incompleteness and inconsistencies [WG05, vL04, GZ05, MZ06, LS99, Ort98].

Incompleteness refers to missing security requirements. The two first parts of this thesis focuses on completeness issues in requirements models for security-critical applications. The first part focuses on completeness issues related to confidentiality requirements. We focus on confidentiality requirements because specifying them formally requires the use of specialized logics that are generally not supported by requirements engineering frameworks. Integrity and availability requirements can be specified in standard temporal logics, which are supported by requirements engineering frameworks. The second focuses on completeness issues related to responsibility assignments.

Inconsistencies refers to requirements prescribing incompatible behavior. Inconsistencies make it impossible to implement some of the inconsistent requirements. As these might be security requirements, this might introduce vulnerabilities in the resulting application. Such vulnerabilities can be identified in the requirements model. The third part focuses on anticipating inconsistencies between requirements when only high-level goals are identified.

The **first part of** this thesis focuses on confidentiality requirements. Unlike integrity and availability, the two other kinds of security requirements from the CIA taxonomy, for example, integrity and availability, confidentiality requirements are not fully supported yet by standard requirements engineering frameworks. Hence, they are not amenable to formal analysis in order to ensure that they are complete and consistent. Confidentiality imposes restrictions on what agents may or may not know about the application over time. Modeling confidentiality requires the expressiveness of epistemic logics to capture agents' knowledge and how such knowledge evolves with respect to the values he can know, to its memory and reasoning capabilities, etc. [BDR04, FHMV95]. A disclosure might be direct if the unauthorized agent has the opportunity to monitor the sensitive information directly, or indirectly, if she can get to know it through observation of other values and reasoning on these values.

We propose a framework for formally specifying and reasoning about confidentiality requirements in the early phases of software development. Our extension makes it possible to automate checks of requirements models against confidentiality claims and discover confidentiality violations at requirements engineering time. Countermeasures must then be found against such violations. These countermeasures yield additional security requirements, which might be confidentiality requirements as well.

The **second part** of this thesis focuses on the assignment of responsibilities for security requirements. Security requirements are assigned to agents who are responsible for enforcing them. For instance, secret holders are responsible for keeping the secret confidential. Some agents might be malevolent or subject to threats. Making them responsible for security requirements can introduce vulnerabilities that might result in security violations.

We propose a technique for ensuring that assets remain protected even in the presence of malevolent or vulnerable agents. The key idea of our technique is to identify trustworthy agents and ensure that the protection of assets is only dependent on whether the trustworthy agents enforce the security requirements they are responsible for. We build on this idea to define a security-specific criterion for requirements completeness. Requirements completeness is about having "enough" security requirements [Yue87]. This

part of the thesis proposes a systematic technique for elaborating all the necessary requirements to ensure that our completeness criterion in met. This technique extends existing goal-oriented requirements elaboration methods [vL01].

The techniques proposed in the two first parts of this thesis focus on completeness issues in requirements models for security-critical applications. Security requirements are often inconsistent with functional requirements [LS99, Ort98]. Such inconsistencies make it impossible to implement an application with the required functionalities and security. In the presence of such inconsistencies, the implementations will either have restricted functionalities for security reasons or the functionalities will allow bypassing the security requirements.

Some recent techniques advocate that one should systematically stay on the "secure side", by ensuring that security requirements have priority over functional ones [BDL06]. This might hurt functionalities and remain undetected until runtime.

In the **third part** of our work, we propose an automated technique for detecting such inconsistencies in a predictive way when elaborating the requirements model through goal-oriented method. Our technique builds on the concept of divergence introduced in [vLDL98]. If a divergence appears during goal-oriented requirements elaboration, then there will be an inconsistency between the requirements that will be derived by this method. Divergences are a way of anticipating inconsistencies at requirements level. We refine and make the definition of divergence further precise. Based on this refined definition we propose an automated technique for detecting divergences.

1.3 Overview of Contributions

This section gives an overview of the contribution for each of the three parts of this thesis.

1.3.1 Reasoning about confidentiality at requirements engineering time

We propose an extended framework for formally specifying and reasoning about confidentiality requirements in the early phases of software development. Our framework has three components:

- **An axiomatic system** captures how the knowledge of unauthorized agents evolves over time. In a way similar to the "Most

Powerful Attacker" hypothesis used in cryptographic protocol analysis, we assume unauthorized agents to be clever and "forgetless" [KMM94, Low96, Bol97, CJM00, Cer03].

Our axiomatic system (1) lists the cognitive capabilities of such most powerful attackers in the context of requirements engineering; (2) borrows from well-known axiomatic systems for knowledge logics and augments them with axioms binding epistemic logic concepts to concepts from security requirements engineering.

- **A catalog of specification patterns** for confidentiality requirements captures different types of confidentiality requirements on state variables. The patterns are expressed in terms of knowledge operators from our axiomatic system. The catalog has a two-dimensional structure according to the allowed degree of approximate knowledge and its timing. These patterns, in the spirit of [DAC98, CP99], considerably facilitate the task of writing specifications with knowledge operators. They can also be used as a means for eliciting confidentiality requirements and making them more precise.

- **A validation method demonstrated by a proof-of-concept tool** checks requirements models against confidentiality claims expressed in terms of such epistemic patterns. It combines bounded temporal model checking and finite instantiation techniques to translate the confidentiality claims into a constraint satisfaction problem [BCC+03, Jac00]. The latter is solved by constraint programming techniques [Sch02b]. The tool generates counter-example traces leading to violation of a confidentiality claim. In such traces, some agent could get knowledge violating the confidentiality claim. This knowledge is gained according to the axiomatic system describing agents' knowledge capabilities. The tool also explains the reasoning performed by this agent to infer the forbidden knowledge.

1.3.2 Assigning responsibilities for security requirements

We propose a technique for ensuring that the protection of assets only depends on whether the trustworthy agents enforce the security requirements or assumptions they are responsible for.

Our technique builds on goal-oriented methods. Goals are prescriptive statements of intent about the system-to-be. Security goals prescribe that system assets should be protected against non-authorized use [HLMN06, vL04]. Security goals are enforced through security requirements, assigned to software agents, and security assumptions, assigned to environment agents [vL03b].

Our technique has four components:

- **A goal-oriented model of trustworthiness** identifies trustworthy agents from high-level security goals. Trustworthiness propagates down to software requirements and environment assumptions. Agents that are trustworthy for a security goal will enforce those requirements or assumptions assigned to it which contribute to the security goal.

- **A security-specific criterion for requirements completeness** captures the key idea of our method in terms of our trustworthiness model.

  A set of security requirements and assumptions is *complete* with respect to a set of security goals if the subset of requirements and assumptions assigned to trustworthy agents enforce all the security goals.

- **A set of local constraints on requirements models** is sufficient to guarantee our completeness criterion. These constraints enable local and compositional reasoning about our completeness criterion. Each requirement or assumption that is not assigned to a trustworthy agent is said to be *untrustworthy*. Untrustworthy responsibility assignments must be resolved through the insertion of countermeasures yielding new security requirements. Three classes of countermeasures have been proposed. They borrow from obstacle resolution tactics [vLL00].

- **A constructive and compositional process** for elaborating requirements model is proposed. This process ensures that the delivered requirements model enforces our completeness criterion. This process builds on the proposed local constraints. This process requires that every responsibility assignment is flagged as trustworthy or not trustworthy. We show that this can be discharged to automated verification tools such as Alloy or PVS [Jac06, SORSC01].

We apply our technique to elicit the security requirements of an e-purse system. Our requirements model is compared to the one prescribed by the European Central Bank (ECB) for such system [ECB03]. This document presents the expectations of the ECB with regard to "*technical security of e-money schemes*". It contains a "*list of security objectives that should be met by e-money schemes in order to cover risks/threats elicited in a comprehensive risk/threat analysis, according to the Common Criteria methodology*" [CC99]. Our approach produces two security requirements that are not mentioned in this document. These requirements are missing because the corresponding threats were not identified. We can produce two attacks on the requirements model based on the fact that these two security requirements are missing.

1.3.3 Detecting divergences in goal models

We propose an automated technique for detecting divergences in goal model. A *divergence* between goals corresponds to situations where some particular combination of circumstances can be found that make the goals logically inconsistent. Such particular combination of circumstances is called *boundary condition* [vLDL98]. The contribution on this thesis is as follows:

- **The formal definition of divergence is made further precise.** One of the three formal conditions defining divergences in [vLDL98] states that the boundary condition must be satisfiable in the domain of the system-to-be. We first show that this condition is too weak and therefore that the definition of divergence is also too weak: some goals can be divergence according to the formal definition without being inconsistent. We extend this condition by mentioning that it must be satisfiable *by the agents of the system that will not have any responsibility related to the enforcement of the divergent goals*. We show that this new definition has the soundness property that divergent goals deliver inconsistent requirements.

  This refined condition should be evaluated when the goals are asserted, without having already identified agents and responsibility assignments. We therefore propose to annotate goals with some information that is both easy to obtain at this stage and from which one can approximate the responsibility assingments. This annotation consists in tagging some of the variables referenced by a goal as "intended to be constrained by the goal". These are said to be the *constrained* variables of the goal [Let01].

  Our refined third condition for divergence states that the boundary condition must be *realizable* by agents of the system not involved in the satisfaction of the divergent goals [PR89b, PR89a, PPS06, Let01]. These agents are approximated by the constrained variables.

- **A technique for detecting divergences** involving propositional goals is proposed.

  A brute-force approach for detecting divergence would be to consider that all subsets of goals might be divergent and to test each of these sets in turn for divergence by testing the existence of a realizable boundary condition. Testing the realizability of a boundary condition can be done using standard techniques [PR89b, PR89a, PR90, PPS06, WDR06].

  The realizability-testing problem is known to be very hard. To avoid testing every combination of goals for divergence, we propose a variant of the realizability-testing technique that enable us to detect all

divergences included within a given set of goals at once. Our variant requires that the set of goal submitted to it enforces some precondition.

Based on this adapted divergence-testing technique, we propose an improvement on the brute-force divergence detection. The sets of goals tested for divergence are the maximal sets of goals enforcing the precondition of our divergence-testing technique. These sets are called "*candidate sets*". We proposed three extensions of our technique in order to keep the candidate sets of goals as few and as small as possible.

The technique has been experimented by means of a prototype tool.

1.4 Organization of the Thesis

This thesis is organized as follows:

- Chapter 2 presents the typical structure of security-critical applications together with two examples of such applications. A state-of the art about security engineering is presented there, with a focus on security requirements engineering.
- Chapter 3 introduces the goal-oriented requirements method and language on which our work is based.
- Chapter 4 presents our extended framework for formally specifying and reasoning about confidentiality requirements in the early phases of software development.
- Chapter 5 presents our technique for ensuring that the protection of assets only depends on whether the trustworthy agents enforce the security requirements or assumptions they are responsible for.
- Chapter 6 presents our automated technique for detecting divergences in goal models.
- Chapter 7 concludes and discusses how the techniques proposed in this thesis can be used by experts in security. It also presents two generic open issues related to requirements engineering and security requirements.

Chapter 2

Engineering Security-Critical Systems

This chapter further introduces the context of this thesis on the basis of two examples. It also presents an overview of the different problems we might encounter in security engineering. More specific related work will be discussed in detail in the next three chapters.

2.1 Two Examples of Security-Critical Applications

We introduce here two examples of typical security-critical applications. They will be referenced throughout this thesis.

2.1.1 An Electronic Payment System

In an e-payment system, electronic money is stored on a smart card-based e-purse installed on bankcards. Money can be transferred from bank accounts onto e-purses at ATM counters. This requires PIN code authentication like for cash withdrawal. Buyers may use their e-purse at dedicated terminals installed in shops. To perform a payment from an e-purse to a terminal, the buyer needs to insert her bankcard in the seller's terminal. The seller enters the amount due in the terminal. This amount is displayed to the buyer so that she can validate it by pressing an "OK" button. The validated amount is transferred from the buyer's e-purse to the seller's terminal. Payment is performed offline and requires no authentication. Terminals are unloaded to sellers' bank accounts via a secured network.

Our model of the e-purse system will concentrate on the part of the

system directly related to the payment operation.

2.1.2 An Electronic Signature System

An electronic signature system enables people to electronically sign official documents. In this system, some people sign documents to show their agreement whereas others need such agreement to act upon the document. The signature sub-system includes an ID smart card (called *e-ID*), an interface between the e-ID and the signer (called *signature box* or *SignBox*, for short), and a PC to which the signature box is connected. The latter has an e-ID reader and a keypad where the signer can dial her pin code. Unlike the PC that might be compromised by viruses, key loggers, root kits, and the like, the signature box is a highly secure device.

2.2 Different Flavours of Security

Security applications cover a range of different problems, namely: software security, socio-technical and organizational security, and physical security.

- **Socio-technical and organizational security** focuses on purely human issues. For instance, in the e-purse system, bank accounts are stored in a central bank office on electronic support. System administrators can have access to this support, hence there are security issues about who may administrate the production systems.
- **Physical security** describes measures that prevent or deter attackers from accessing a facility, resource, or information stored on physical media. It ranges from locked doors with physical access control to smart card tamper-resistance.
- **Software security** refers to the effort of creating a secure computing platform, designed so that users or programs can only perform software-defined actions that are allowed. This involves specifying and implementing security policies. The actions in question can be reduced to operations of access, modification, and deletion. Computer security can be seen as a subfield of security engineering, which looks at broader security issues in addition to computer security.

Here, we focus on software security.

2.3 Structuring Security-Critical Software Applications

We further detail here the layered organization of security-critical software application introduced briefly in Section 1.1 [Win98], this organization is illustrated on the purse and e-signature systems.

2.3.1 Cryptographic Layer

The problem at this layer is to provide primitive techniques for enforcing data confidentiality, authentication, watermarking, etc.

An example of primitive building block for achieving confidentiality is a pair of functions f and g such that: (1) any digital data d can be encoded and decoded by applying f and g: $f(g(d)) = d$ (2) given the encoding $g(d)$, it is impossible or at least very difficult to recover d without knowing f. An encryption technique and a key generally implement such functions. The key is a list of bits that is used as a parameter of the encryption technique. Using keys allows encryption techniques to be made public and standard. There are several standard encryption mechanisms known as RSA, DES, 3DES, AES, etc [RSA78, oST, DR02].

Another primitive is the one-way hash function. This building block is made of a function f such that, given the encoding $f(d)$, it is impossible or at least very difficult to recover d.

The security assurance about cryptographic primitive relies on complexity theory. In the case of encryption functions, the principle is that, recovering a data d from its encoding $g(d)$ without knowing the necessary key would require a computation time proportional to, for example, the exponential of the key length. Some agreement exists about the strengths of the available solutions with respect to the key length [LV00, Len04, ECR06, NIS06, NSA05].

A well-known pitfall in cryptography is to rely on the confidentiality of the encryption mechanism to claim that the encrypted data cannot be recovered without the proper keys or without the knowledge of the encryption mechanism. This is widely accepted not to provide enough security assurance. A general principle is widely accepted in cryptography about this matter. It is known as *Kerkhoffs' principle* [Ker83]. The principle states that the security of a system should be based on the strength of its protection mechanisms, not on the obscurity of its design.

2.3.2 Protocol Layer

At the protocol layer, the problem to solve is to provide secured communication services for application-specific purposes. For instance, in the case of the e-purse system, how could a terminal and an e-purse ensure that they are connected to a genuine e-purse and terminal, respectively?

Such functionalities are proposed by security protocols. A security protocol is specified by the security properties it enforces, such as mutual authentication, as suggested here above. It is implemented by a set of messages to be exchanged between the communicating principals. These messages generally contain data encrypted or hashed through the techniques provided by the cryptographic layer.

Protocols range from general-purpose protocols such as mutual authentication, to application-specific protocols such as protocols for credit card payment clearing [BGH+00].

Protocols might be flawed in the sense that they do not enforce their specification. For instance, defeating a mutual authentication protocol means that the attacker is able to impersonate another principal. The attack might consist in intercepting, modifying and injecting messages into an ongoing protocol session.

Efficient techniques have been proposed for testing protocols against their specifications and for deriving protocols from specifications [Bol97, CCC+04, CJM00, Wor97, Low96, DY81]. These techniques rely on a model of what the attacker is able to do, for example, intercept a message, decode encrypted information whenever he has the necessary cryptographic keys, and inject new messages. These capabilities define an attacker model. The general agreement is that the modeled attacker should have as many capabilities as possible in order to ensure that if no attack can succeed given these capabilities, the protocol is indeed "secure" against real attacker with potentially more restricted capabilities [KMM94, Low96, Bol97, CJM00, Cer03]. Such an attacker model is generally known as the *Most Powerful Attacker Model*.

These techniques generally rely on specialized logics such as *BAN logics* [BAN89]. BAN logics support specialized constructs with associated meaning. For instance, the construct $\{X\}_K$ represents the value X encrypted under key K. Ban logics come with rewrite rules. These rules capture the capabilities of the attacker.

An important issue at the protocol layer is that the protocol used in an application must be adequately chosen with respect to the application-specific security requirements. This is known to be a difficult problem because understanding the properties of a cryptographic protocol requires high

expertise, generally not available to application specialists [Mea03, And93].

In the e-signature system, the signature of electronic documents is generally implemented by means of cryptographic hash of document content such that (1) the electronic signature can be checked by any user having the document and the signature (2) the signature can be generated by the signer only. The signature itself can be viewed as a message of a cryptographic protocol; it is made of several data items (document content, signer's identity, etc) assembled through cryptographic functions.

2.3.3 Language and System Layer

At the language and system layer, running implementations of security-critical applications are provided. For software applications, the implementation is performed by producing some source program code and compiling it into binary executable form. The produced program must enforce a given set of requirements.

Program code implementing security-critical applications can be subject to attacks. At this layer, the objective of an attack is to make the program behave unexpectedly and, typically, in a way that does not match the security requirements it is supposed to enforce. For instance, confidential data such as credit card numbers or commercial data might be disclosed via such attacks even though some access control protection has been set up.

The most famous attack at this layer is the "buffer overflow" attack consisting in sending data containing malicious code and managing that the malicious code is executed by the recipient of the data. It mainly arises because of implementation weaknesses. Buffer overflows can either be detected at compile time via static analysis techniques or at run time via code instrumentation [PS05, GPSAK06].

Beside these attacks, one should ensure that the source code implementation matches the expected behavior. Dedicated code analysis techniques have been proposed to check a program does not allow users to access data they should not access [DD77].

The executable code runs on an operating system. The latter might also be subject to attack and subverted by similar techniques. Once the operating system is subverted, every application running on it can also be subverted even if they are robust against attacks. As a result, operating system protection has gained a lot of importance. Techniques for preventing such attacks include firewalls that restrict the network connections of the protected system to the outside world.

Intrusion detection systems are the final line of defense against attacks at this layer [Den87]. The purpose of such systems is to detect that an attack

is currently ongoing in order to provide some adequate reaction. The latter might be shutting down the software system, isolating the system from the network, etc. The detection mechanism is typically driven by analysis of audit traces produced by the operating system. *Honeypot* techniques are another way of achieving the same detection goal. The principle is that the attacker is attracted to act in some predefined way [Spi02]. This is typically achieved by providing him with targets to attack and making these targets look as interesting as possible or as big as possible [VBBC04]. Depending on the type of target, the intrusion is detected whenever these targets are attacked or simply contacted through a network connection.

For instance, in the electronic signature system, the signature box device must be robust against subversion through buffer overflow attacks. In the purse system, the terminal should be robust against buffer overflow attacks initiated by potentially malevolent faked e-purses.

2.3.4 Application Layer

This section reviews the application layer. It is presented with more details than the other layers because this thesis focuses on security at this layer exclusively.

2.3.4.1 Defining Application-Layer Security

At the application level, the problem is to elaborate application-specific security requirements to be implemented in the system-to-be so that the application remains dependable in the face of malice [And01]. This is also referred as "security mechanism" instead of "security requirement". The security requirements altogether are sometime referred to as the *security target*.

Dependability is that property of a computer system such that reliance can justifiably be placed on the service it delivers [Boa00]. The object of this reliance is application-specific. It can be defined by a set of security properties that the system-to-be should satisfy, also known as *security goals*.

The security engineering community commonly agrees that most of these security goals can be classified in the C.I.A. taxonomy presented in Section 1.1 [CC99, Amo94, Kem03]. This taxonomy does not cover all kind of desirable properties. For instance, in a service system, one can find the two following security goals: (1) to get paid for a provided service and (2) not to pay for an non-requested service. These goals do not fit into the C.I.A. taxonomy. As a consequence, several sources have argued in favor of an extension of this taxonomy to include, for example, accountability,

privacy, trust, non-repudiation , authentication [CPP06, AEPA01, YL00]. These additional properties are defined as follows:

- **Accountability** requires corresponds to the availability and integrity of a set of meta-information concerning the existence of an operation, the identity of the person who realised the operation, the time of the operation, etc. [Boa00]
- **Privacy** is confidentiality with respect to personal data, which can be either "information" (such as the content of a registration database), or "meta-information" such as the identity of a user who has performed a particular operation, or sent a particular message, or received the message, etc. [Boa00] It also includes the purpose for which information is disclosed and what can be done with it [AEPA01].
- **Trust** requires that the system should be such that its users feel confident in it [YL00, Sin06].
- **Non-repudiation** corresponds to the availability and integrity of some metainformation, such as creator identity (and possibly time of creation) for nonrepudiation of origin, or such as reception and receiver identity for non-repudiation of reception [Boa00] .
- **Authentication** requires to have some assurance of authenticity. *Autenticity* is the property of being "genuine". For a message, authenticity is equivalent to integrity of both the message content (information integrity) and of the message origin, and possibly of other meta-information such as time of emission, classification level, etc. (meta-information integrity). Similarly, a document is authentic if its content has not been altered (information integrity) and optionally if the declared author is the real author and not a plagiarist, if the publication date is correct, etc. (meta-information integrity). Likewise, an alleged user is authentic if the declared identity is the real identity of that person [Boa00].

At this level, a *security failure* is a state of the system that is inconsistent with a system's security objectives [LNIJ04]. Security failures are made possible because of vulnerabilities. A *vulnerability* is defined as the condition of a system that, in conjunction with an attack, can lead to a security failure [Boa00, CC99]. For instance, there is a vulnerability if a necessary security requirement is not enforced. This might occur either because it has not been identified or because it is not enforced by the implementation.

Requirements engineering is concerned with the elaboration, specification, analysis and documentation of application-specific requirements. The purpose of requirements engineering is to elaborate complete, consistent,

precise and adequate requirements for specific applications. All these properties are obviously desirable for security targets as well.

- *Completeness* ensures that all the requirements were identified [Yue87]. In the context of security requirements engineering, a missing security requirement is a vulnerability to be exploited by attackers.
- *Adequacy* ensures the specified requirements define the real needs for a new system.
- *Precision* ensures that the requirements do not have multiple interpretations of interests making it true [vL00]
- *Consistency* ensures that requirements can be enforced together. No implementation can enforce an inconsistent set of requirements [vLDL98]. In the presence of inconsistencies involving a security requirement and a functional requirement, the implementations will either have restricted functionalities for security reasons or the functionalities will allow bypassing the security requirements.

Security requirements engineering is an area that has received little attention so far [vL04, DS00]. Textbooks on security engineering argue that this is an important issue but no technique is widely available [And01, Bis04].

The problem of elaborating security targets can be roughly divided into two related sub-problems:

- Identifying security requirements and analyzing their completeness , adequacy, precision and consistency.
- Deriving security mechanisms from identified security requirements.

We should also care about the identity of agents that are responsible for enforcing security requirements [HLMN06, GMMZ05a].

2.3.4.2 Characterizing Good Approaches

Ideally, a good approach towards the elaboration of security requirements should have the following characteristics [vL04]:

- **Early deployment** In view of the criticality of security requirements, the technique should be applicable as early as possible in the RE process, that is, to declarative assertions as they arise from stakeholder interviews and documents (as opposed to, for example, later state machine models).

- **Incrementality** The technique should support the intertwining of model building and analysis and therefore allow for reasoning about partial models.
- **Reasoning about alternatives** The technique should make it possible to represent and assess alternative options so that a "best" route to security can be selected.
- **High assurance** The technique should allow for formal analysis when and where needed so that compelling evidence of security assurance can be provided.
- **Security-by-construction** To avoid the endless cycle of defect fixes generating new defects, the RE process should be guided so that a satisfactory level of security is guaranteed by construction.
- **Global approach** The technique should integrate the elaboration of security requirements with the elaboration of functional requirements so as to allow for interaction analysis [vLDL98, Rob89].

2.3.4.3 Early Times: Good Practice Guidelines

In the early time of security engineering, **good practice guidelines** were the only methodology for solving these two problems together. The eight most famous principles have been proposed by Saltzer and Schroeder [SS74]. They are as follows:

- **Economy of mechanisms:** The system should be sufficiently simple and small to be verified and implemented.
- **Fail-safe defaults:** In case of uncertain situation, the correct behavior of the application should lie on the secure side. For instance, access control should be denied by default and explicitly allowed when necessary.
- **Complete mediation:** Every access to every object must be checked for clearance.
- **Open design:** The system should be open for scrutiny by the community. Users should be able to convince themselves that the system meets their security requirements. The purpose of this principle is to involve a large amount of people in the design of the system and to hopefully cover all possible flaws. Notice that this principle borrows from Kerkhoffs' principle in that the security of an open design cannot rely on the obscurity of the design (cfr. Section 2.3.1).

- **Separation of privilege:** Where feasible, a protection mechanism that requires two keys to unlock it is more robust and flexible than one that allows access to the presenter of only a single key.
- **Least privilege:** An agent should only have the rights necessary to complete her task.
- **Least common mechanisms:** Mechanisms used to access resources should not be shared.
- **Psychological acceptability:** The user interface should be designed for ease of use, so that users routinely and automatically apply protection mechanisms correctly.

All these principles originate from the very beginning of security engineering (1883 for Kerkhoff's principle). They still apply now [Kem03, And01, Bis04].

2.3.4.4 Browsing through the C.I.A. taxonomy

Identifying the desired security properties of a system requires a deep understanding of the business of the system [And01]. A first approach for identifying them is to browse through the C.I.A. taxonomy. The security properties from this taxonomy that are relevant to the considered system can be added and specialized in application-specific terms [And01]. In order to facilitate this process, several textbooks propose a refined version of the C.I.A. taxonomy, together with examples of use [CNYM00, CC99].

For instance, the NFR method distinguishes between two different kinds of confidentiality requirements: internal confidentiality and external confidentiality. *Internal confidentiality* refers to the confidentiality of information items residing in the system. *External confidentiality* refers to the confidentiality of information items residing outside the system. In this context, the word *system* refers to a central office managing a possibly distributed set of agents.

2.3.4.5 Reasoning About Attacks and Threats

Another approach for identifying security requirements is to question about the harm that can be caused to the system-to-be. These are *threats*. A *threat* indicates the potential for abuse of assets [LNIJ04, And01, DS00]. Once the threats are identified, desired security properties can be derived from them, by negating the harm caused by the threat [And01, Sch97]. For instance, the industrial *common criteria methodology* suggests that security mechanisms should be elaborated to cover threats identified in a comprehensive

threat analysis [CC99]. In this methodology, desired security properties are therefore indirectly captured into threats.

An *attack* is a scenario for threat instantiation. Reasoning about attacks can also help to identify threats. Reasoning about attacks has been successfully applied in cryptographic protocol analysis [Low96, Sch97, Sch96, Bol97, CJM00, KMM94].

This approach relies on the fact that an attack can occur if there is a vulnerability; each identified attack therefore reveals a vulnerability. Vulnerabilities are generally resolved by adding appropriate countermeasures.

The difficulty of such attack/threat-oriented approaches is to be sufficiently exhaustive in the attack/threat identification phase. This is critical because a non-envisioned attack might exploit a vulnerability that is not identified and therefore not resolved. As a result, the security requirements might be incomplete, thus vulnerable to non-envisioned attacks [Sch97]. The countermeasures added to a model might introduce new vulnerabilities [And01, vL04, HMLN06]. Several techniques and heuristics have been proposed in order to facilitate the threat identification process and head towards completeness. We overview some of them here.

In the ideal case, a complete operational model is available and the security goals are available. In this case, all attacks can be discovered through symbolic model checking techniques and represented through *attack graph* models [SJW02]. However, operational models are seldom available and security goals are not clearly identified.

Threats can be identified by **browsing through the assets** involved in the system [HMLN06]. The principle is to identify the harm that can be caused to each of the assets involved in the system in turn.

In order to facilitate the attack identification phase, *attack trees* have been proposed in [Amo94, Sch00]. An attack tree is a tree in which the nodes represent attacks or steps of attacks. The root node of the tree is the global goal of an attacker. Children of a node are refinements of this goal, and leafs therefore represent attack steps. A refinement can be conjunctive (aggregation) or disjunctive (choice). A conjunctive refinement means that the children goals are different steps towards achieving the refined goal. A disjunctive refinement means that the children are alternative ways of achieving the refined goals. This representation offers support for a systematic exploration of the attacks on the system-to-be. Like fault trees, attack trees also enable analysis such as cost and risk analysis [MO05]. This is very useful when reasoning about socio-technical and physical aspects.

A variant of the attack-tree notation specialized for software security has been proposed in [vL04]. In this variant, nodes represent conditions on the system that an attack might intentionally manage to make true. A

set of anti-goals refines an anti-goal if the refining ones entail the refined anti-goal. Such anti-goal trees can be explored upwards, in order to identify the rationales behind the attack and the possible intentionalities of potential attackers. Such intentionalities and rationales might serve as a new basis for envisioning alternate attacks on the system-to-be.

A variant technique for finding attacks has been proposed in the context of application running within a social setting [LYM03]. In this variant, each human actor interacting with the software system is considered in turn as an attacker. For each attacker identified, the capabilities and interests of the attacker are combined with those of the legitimate actor. Attacks causable by this agent are then easier to identify. The *abuse case* approach also suggests to identify potential attackers first and then identify attacks on the system caused by them [MF99]. These attacks are represented through a variant of the use case notation called abuse case. These are then refined and expanded to model what the security mechanisms of the system need to defend against.

Through the problem frame notation, *abuse frames* make it possible to express a security threat explicitly by relating undesirable phenomena to a diagrammatic description of the system context. The technique facilitates the structuring of a partial system for early threat analysis when the requirements of the envisioned system are being elaborated [LNIJ04].

Another technique heading towards the exhaustivity of identified attacks is by reusing **attack libraries**. [And01]. The principle is to propose a set of standardized attacks and to systematically test these attacks on the system-to-be. *Attack patterns* are a form of generalized attack libraries where one pattern capture a set of attacks. For instance, the man-in-the-middle attack and the flooding attack are well-known attack patterns that can be instantiated to many different situations [And01]. The principle of building a catalogue of known threat tree patterns for documentation and reuse through instantiation is nicely illustrated in [MEL01].

2.3.4.6 Refinement-Based Approaches

Another aproach for deriving security mechanisms from security properties is to reason on an intermediate level of abstraction between high-level security goals and the security requirements to be implemented. In an e-signature system, a high-level security goal might be that a document should not be considered as electronically signed by a user if she had never agreed on this document. A security requirement from e-signature system might be that the e-ID should only provide a signature for an electronic document if it is given the correct PIN code together with the data to sign.

Historically, the principle of introducing an intermediary level of abstraction appeared in security policies. A *security policy* is a succinct statement of a system's protection strategy (for example, "each credit must be matched by an equal and opposite debit", "all transactions over $1,000 must be authorized by two managers") [And01]. The principle is therefore to derive a **security policy** from the security goals and to derive security requirements / mechanisms from the derived security policy.

The word *security policy* has been considerably overloaded [DS00]. Some sources define security policy as security target [And01, DDLM00]. Such "low-level policies" can directly be enforced by off-the-shelf software component such as the framework available for policies defined in the Ponder language [DDL+02]. *Ponder* is a formal language for defining security policies [DDLM00]. Ponder is a declarative, strongly typed and object-oriented language supporting concise notations for declaring access control policies and obligation policies. The ponder software framework can be connected to java software application, operating system-level access control as well as database systems.

Protection profiles are a form of intermediate-level security policies. It is defined as an implementation-independent set of security requirements for a category of security-critical system that meet specific consumer needs. It has been defined in the context of the common criteria [CC99].

To come back to the "intermediate level policies" we consider here, it turns out that lots of these security policies are very similar in mind to each oher. Standard models of security policies have been propozed to address standard security properties. The most famous ones are the *Bell-la-padula*, *Biba*, *Lipner*, *Clarck-Wilson* and *Chinese Wall* models [BL73, Bib75, Lip82, DW87, BN89]. The Bell-La-Padula security policy model is dedicated to enforce data confidentiality while the Biba, Lipner and Clarck-Wilson policy models focuses on integrity. The Bell-la-Padula model is presented here as a typical example of security policy model.

In the Bell-la-Padula policy, data are called objects; human agents and software processes are called subjects. Objects and subjects are assigned a classification level. Classification levels are totally ordered. The desired property is that no subject should get to know an object whose classification level is higher than the subject's. In order to achieve this property, two rules must be enforced:

- **No Read Up**: A subject cannot access object whose classification level is higher than the subject's.
- **No Write Down**: A subject cannot write to an object whose classification level is lower than his own.

A schematical example of situation is represented in Figure 2.1. There is one subject and two objects. The classification levels (1, 2 and 3) are delimited by the horizontal lines. The four possible accesses that the subject can perform on the two objects (Read/Write Up/Down) are represented by data-flow arrows and the two forbidden accesses are crossed out.

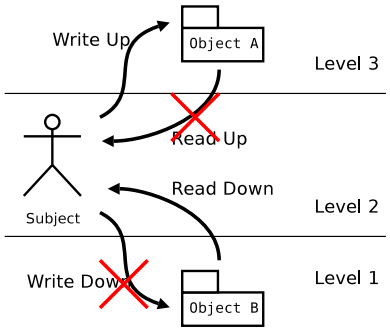

Figure 2.1: Schematic representation of the Bell-La-Padula security policy model

These security policy models are very generic and might require substantial effort in order to properly derive the security mechanisms from them within a given application [VGV92]. In order to build up an application-specific security policy from generic security policies, composition rules have been proposed [GQ94, GQ96].

Another family of refinement-based approaches is goal-oriented approaches. Goal-Oriented methods are well established for eliciting, representing, specifying and reasoning about requirements [vL01]. The basic idea of these methods is to derive software specifications in a constructive way from high-level goals. Goals are desirable properties of the system. They are refined from high-level concerns to requirements, which can be assigned to agents. These methods offer powerful techniques for conflict and obstacle detection and resolution [vLDL98, vLL00]. The KAOS goal-oriented method is presented in full detail in Chapter 3.

The NFR method is a goal-oriented method for reasoning about non-functional security requirements [CNYM00] In this context, non-functional goals or requirements cannot be stated as enforced or not enforced, mainly because they cannot be formally defined. The main idea is to reason about the interaction between goals. Such interaction may be positive or negative. In order to provide efficient guidance, the method proposes a very rich taxonomy of security goals together with standard relations that can be found between them, for instance; between confidentiality and authentication goals.

2.3.4.7 Socio-Technical and Organizational Security

Socio-technical and organizational security requires specialized requirements engineering techniques in order to reason about the concepts of this area, such as trust, delegation and ownership.

Trust modeling has often been considered in computing science as a basis for making decisions with incomplete knowledge about agents. Trust management systems help making security-related decisions based on trust models [CFL+97, GS02].

Trust models range from tree models for credentials such as the one used in PKI systems [Mau96] to models based on deontic logic or finer-grained models featuring levels of trust. Such levels can be either numeric or lattice-based. Numeric levels can be positive for representing levels of trust and negative for representing distrust [Man00, Cap04, Dem04].

These models differ by their purpose and by the aspects they capture, for example, competence, recommendation, history, etc. They also provide some axioms on how to combine the various features they capture. A level of trust can be derived from credentials or historical data and compared with some fixed threshold or with results of risk analysis for decision-making.

Most models come with a specific definition of trust that fits the feature it captures. This results in a large number of different definitions [nin, TL04]. Several models are based on trust matrices or analogies with human trust [Man00, Cap04].

Two broad classes of definition of trust can be distinguished. We give a typical instance of each class.

- According to the national security agency of the United States, we *trust* a component if we give it the ability of breaking security. A *trustworthy* component is one that would not break security, should it be trusted [And01].
- According to the Cambridge definition [cam], you *trust* a component if you believe it to be trustworthy. A *trustworthy* component is one that will do what it says.

Variants around these definitions refer to belief quantification, trustee competence, trustee honesty, truster propension to trust agents in general, historical data, and so forth.

The *I* framework* and the *Tropos framework* propose agent-oriented approach to requirements engineering centering on the intentional characteristics of the agent [Yu95, BPG+04].

The I* framework supports concepts for modeling intentional properties of agents, such as goals, beliefs, abilities, and commitments. Dependencies between agents give rise to opportunities as well as vulnerabilities. Networks of dependencies are analyzed using a qualitative reasoning approach. Agents consider alternative configurations of dependencies to assess their strategic positioning in a social context.

The *Secure Tropos framework* is an extension of the I*/Tropos frameworks that supports specialized concepts for reasoning about security, such as ownership, delegation and social dependency for defining the obligations of actors (dependees) to other actors (dependers) [GMMZ04a, GMMZ04b]. A dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource. The concepts of the secure Tropos framework have been refined in order to capture more accurately the various situations appearing in socio-technical models [GMMZ05a]. The refinement includes a distinction of the notions of delegation of execution (at-least delegation) and delegation of permission (at-most delegation), also the distinction of the notions of trust of execution (at-least trust) and trust of permission (atmost trust). An formal axiomatic system has been proposed to define these various concepts and the relation between them. For instance, if an actor owns a resource, he has the permission to access it. These axioms are formalized in datalog and enable automated formal reasoning on these concepts in order to detect undesirable situations, such as an actor that has the permission to access a resource owned by another actor who does not trust him. The Tropos and Secure Tropos methodologies are supported by a CASE tool [GMMZ05b].

2.3.4.8 Reuse

Throughout this section, we have presented several forms of reuse, namely: through well-known attack patterns or attack libraries, and through the use of standardized policy models. Browsing through the CIA taxonomy can also be seen as a form of reuse.

Several proposals go one step further in this direction and prone the complete reuse of security requirements. These proposals argue that sets of security requirements should be similar between several applications and can therefore be reused to diminish the occurrence of errors in security targets [SFO03, Fir04, Gar99b, Gro04].

As a consequence, the same authors suggest that requirements models should be written so that they are easily reused [Fir03].

2.3.4.9 Peer Reviewing

A common approach in all engineering areas is to ask for product inspection and reviews. This approach is stressed with more importance for engineering problems where no clear assessment can be done about the quality of the result, as for security requirements engineering.

Many authors therefore stress the need for such peer reviewing for security requirements models, with the underlying principle that the requirements elaboration phase should deliver a correctness argument for the elaborated requirements model [ZJ97, Jac01, BDLN05]. This is also stressed by the *open design* principle presented in Section 2.3.4.3.

The misuse case and abuse case notations have been proposed in order to facilitate stakeholder interaction, which is a form of peer review [SO05, MF99]. They are based on the "use case" notation, borrowed from UML standard. Many stakeholders feel comfortable with this notation, hence it facilitates stakeholders interaction [WPJH98].

Some proposal go one step further and advocate that the principle of peer reviewing should be extended into parallel engineering [And99]. The principle is to hire independent security experts or groups of experts to derive the security target from a given set of desired security properties. The security target resulting from this process can be taken as the "supremum" of the several elaborated security targets.

2.3.4.10 On the Criticality of Inconsistencies

Some authors claim that one of the main classes of defects in requirements specifications is inconsistency [GZ05, MZ06, WG05].

For instance, in the e-purse system, we have a requirement stating that no payment should take place if the buyer does not validate the amount by pressing the "OK" button placed on the terminal. An inconsistency would arise if there was a functional requirement prescribing that the payment must occur within some conditions and if these condition can arise while the "OK" button has not been pressed.

Inconsistencies make it impossible to design and implement a system that respects its specification. In some cases, inconsistencies are arbitrarily resolved by the programmer during the implementation of the system [GZ05, MZ06, WG05]. In the e-purse, the developer must chose between respecting the functional requirement or the security requirement.

This means that one of the inconsistent requirements will be preferred over the others, usually without performing an in-depth analysis of the consequences, and without notifying the relevant stakeholders. Even worse, un-

detected inconsistency can lead to incorrect and unreliable systems, whose faults are only discovered when it is too late: during operation. The choice presented above might not be explicitly made by the developer because he might not be aware that he is actually making such a choice while developing the application.

As a result, a common attack on an existing system is to put it in a state where at least one requirement must be violated because of an inconsistency. The violated requirement might be a security requirement, in which case, there is a vulnerability.

Inconsistencies are also encountered in security policies. Security policy might include negative and positive prescriptions. A positive prescription states that some action must be taken in some circumstances. A negative prescription states that some action requires that some condition is met before being taken. It might happen that a positive and negative prescription are inconsistent with each other, under some circumstances [LS99].

Two main schools have emerged with respect to the treatment of inconsistency in software specifications [GZ05]. The first one proposes techniques and tools to ensure that inconsistency is not present in the specification [vLDL98] As this is difficult to obtain in practice, the second school proposes instead that inconsistency may be tolerated, and resolved at a later stage, typically, at run time [LS99]. For example, inconsistent requirements might be included into a security policy and resolved at run time, when the inconsistency is encountered.

Several techniques for resolving inconsistencies within security policies at run time have been proposed [LS99]. They are as follows:

- the negative requirement (that is, the one prescribing that something must be forbidden) has the priority,
- the requirements all get a static priority and in case of conflict, the one with a lower priority might be violated if it is a necessary condition for ensuring that higher priority requirements stay unviolated,
- the most specific requirement has the priority, this requires that the specificity of a requirement with respect to a state of the system can be measured

The weakness of the second school is that the resolution might lead to unexpected behavior at run time. For instance, functionalities might be blocked in case the first technique is applied or the specificity of a security requirement might not be the accurate ground for a conflict resolution. The purpose of the techniques described here for resolving inconsistencies at run time is to make the behavior as expected as possible. Another concern is about system mixing both security aspects and safety aspects. What if a

safety and a security requirement are inconsistent? Or what if two security requirements are inconsistent?

2.3.4.11 Global Approaches

What we call global approaches is a combination of several of the techniques presented in this section, where for instance the result of one of the techniques is used as the input to another one. The Tropos and I* methodology propose such a process based on their specific techniques [LYM03, Yu95, BPG+04].

The Security Quality Requirements Engineering (*SQUARE*) Methodology consists of nine steps that generate a final deliverable of categorized and prioritized security requirements [MHI05]. The steps include the interaction steps between stakeholders and requirements engineers, risk analysis, threat identification, inspection, etc.

Another framework has been proposed in [HMLN06]. This framework is based upon the construction of a context for the system and satisfaction arguments for the security of the system. One starts with enumeration of security goals based on assets in the system. These goals are used to derive security requirements in the form of constraints. The system context is described using a problem-centered notation, and then this context is validated against the security requirements through construction of a satisfaction argument. The satisfaction argument is in two parts: a formal argument that the system can meet its security requirements, and a structured informal argument supporting the assumptions expressed in the formal argument [BDLN05]. This framework uses Jackson's Problem Frames approach for specifying the relationship between security requirements and the specification of software behavior [Jac01].

Finally, the SCR method proved useful for analyzing models of security-critical systems and proving the compliance of the model with given security requirements elaborated through attack-based approaches [scr06].

2.4 Discussion

This thesis focuses on security at the application layer exclusively. At this layer, we focus on completeness and consistency issues arising in requirements models for security-critical applications. Completeness and consistency issues are particularly crucial when elaborating security requirements because application-level vulnerabilities arise from such issues.

Most of the approaches presented in the previous sections for engineer-

ing security requirements do not support or guarantee a clear criterion for requirements completeness for the various reasons discussed below:

- Browsing through the CIA taxonomy and selecting adequate and relevant requirements does not guarantee that all necessary requirements are identified. There are two reasons for this. The first one is that a taxonomy does not include domain-specific concepts such as, in the case of the e-signature, the concepts of signature, signer, e-ID, etc. Instead, one might find standard terminology, such as "authorized agent", "integrity", etc. Such concepts must be further defined into application-specific concepts. This instantiation is as critical as identifying the requirements from scratch because, for instance if one identifies too many authorized agent in the application domain, the requirements model might be vulnerable. The second reason is that the taxonomy must be rich enough to capture all possible classes of requirements that might be included in a requirements model.

- Good practice guidelines partially rephrase the need for requirements completeness through the "complete mediation principle". They however refer to concepts that must be made precise. For instance, the complete mediation principle refers to a "check for clearance", that is: unauthorized agent should not have access to protected resources. These concepts must be defined in term of concepts from the application domain, like in the browsing approach discussed here above.

- Identifying attacks and proposing security requirements as countermeasures to them does not guarantee that all relevant security requirements are identified because it relies on the fact that all relevant attacks are identified. The techniques presented in Section 2.3.4.5 for identifying attacks on a requirements model do not guarantee that all the attacks are identified.

- Reuse of existing solutions and peer reviewing are standard solutions used in all areas of engineering when ones want to solve a difficult problem. It generally provides a better solution to the problem than the one produced from scratch by a single engineer. However, in this case, it cannot guarantee that the resulting application is of good quality because it relies on the quality of an existing requirements model and its tuning to the specific context of the new application, or it relies on the competence of the peers. These are not formal proofs of completeness. Yet these approaches are very important techniques in all areas of engineering.

- Refinement-based techniques such as policy-based techniques in principle can produce some form of completeness because a policy achieves

some desired security property. For instance, the Bell-La-Padula security policy guarantees that the objects are kept confidential through no-read-up and no-read-down requirements.

This thesis proposes three techniques for reasoning about requirements models. Reasoning about models notably enables us to provide a clear criterion for requirements completeness, with systematic requirements elaboration methods. The price to pay is that models might be inaccurate to the real world. As a consequence, reasoning about models and expert knowledge are complementary approaches in the elaboration of requirements models for security-critical application. This issue is discussed in Section 7.2.

This research builds on goal-oriented approaches featuring a refinement-based process for elaborating requirements models. Our approach thus belongs to the class of refinement-based approaches discussed in Section 2.3.4.6. A goal-oriented requirements engineering method is presented in Chapter 3. In this thesis, we build on this method although other methods could have been chosen as well (see the discussion in Section 3.1).

The objective is to provide high-assurance systems. We therefore head towards formal approaches, like the one described in Chapter 3. Our scope is voluntarily restricted to software security (see Section 2.2) because the behavior and the environment of software applications can be formalized in an easier way than physical and socio-organizational systems.

Goals already provide a *completeness criterion* for the requirements specification - the specification is complete if all stated goals are met by the requirements [Yue87]. However, this criterion does not care about the identity of agents that are responsible for enforcing security requirements, which has been presented as an important issue in security engineering in Section 2.3.4.1. We use this criterion as a basis for defining an extended criterion for requirements completeness in Chapter 5. Our criterion extends the goal-oriented one in that it also takes into account that agents might not enforce the requirements they are responsible of either because they are malevolent or too vulnerable to attack.

Goal-oriented methods however do not support the concept of confidentiality requirements, similarly to other formal methods for reasoning about requirements models. We first need to formally define requirements if we were to prove any completeness criterion. This is why Chapter 4 first extends the expressivity of these languages with a technique for defining and reasoning about confidentiality requirements.

Goals provide the basis for identifying the origin of inconsistencies within requirements models [vLDL98]. It has been shown that inconsistencies between requirements originate from inconsistencies between goals [Rob89]. In Chapter 6, we extend these bases in order to provide automated tool support

for detecting inconsistencies between goals.

In addition to these qualities, goal-oriented approaches mostly match the characteristics presented in Section 2.3.4.2 that describe the desired properties of a good approach towards the elaboration of security requirements:

- **Early deployment** Goals have proven to provide an appropriate support for early time and even for interviews and documents [vL01].
- **Incrementality** The technique supports the intertwining of model building and analysis and therefore allow for reasoning about partial models [PMR+04].
- **High assurance** The method we have chosen provides a sound basis for defining goals formally, hence provides the necessary high assurance [Let01].
- **Security-by-construction** Refinement-based approaches enables us to produce requirements model with some desired properties, by construction. For instance, goal-oriented approaches produce requirements model that enforce high-level goals. These might be security goals.
- **Global approach** Goal-oriented methods integrate the elaboration of security requirements with the elaboration of functional requirements and includes techniques for interaction analysis [vLDL98, Rob89].

Chapter 3

Goal-Oriented Requirements Engineering

This chapter describes the goal-oriented requirements engineering method and parts of the KAOS language supporting this method. This chapter heavily borrows from [Let01, Dar95, vL03a].

Section 3.1 presents the reasons why we have chosen to build this thesis on the KAOS framework instead of other goal-oriented methods. Section 3.2 present the KAOS goal-oriented requirements specification language. It is illustrated on the e-signature system. Section 3.3 presents the goal-oriented requirements elaboration method. It is illustrated on the e-purse system. Section 3.4 summarizes this chapter.

3.1 Why the KAOS Goal-Oriented Method?

The main frameworks for goal-oriented requirements engineering are:

- **The KAOS framework** is mostly concerned with the elicitation, elaboration and analysis of requirements models through goal-oriented analysis [DvLF93, DFvL91, Let01, vL03a]. The requirements and the goals are defined in temporal logic.
- **The NFR framework** is mostly concerned with the evaluation and selection of alternatives designs with respect to qualitative 'non-functional' goals such as usability, performance, accuracy, security, etc. [CNYM00, JCN92].
- **The i* Framework** is mostly agent-based framework for modeling social interactions and strategic relationships [YL00, Yu95]. However,

it has also been applied to requirements elicitations and comprises the concept of goal.

- **The Tropos framework** and its variants *Secure Tropos* and *Formal Tropos* are generaly considered as the successorts of the NFR and i* frameworks [BPG+04, GMMZ04a, GMMZ04b]. It is mostly devoted to the analysis of socio-technical systems.

In this thesis, we use the KAOS framework because

- This framework explicitly represents the concepts of agents and supports formal definition of requirements. These are the concepts we need to reason about in our technique for reasoning about confidentiality requirements in Chapter 4. Although originally developed in the context of the KAOS method for goal-oriented requirements engineering, our technique for reasoning about confidentiality requirement was designed for use within other declarative frameworks.
- This framework explicitly represents goal refinements and agent responsibilities. These are the concepts we need to reason about in Chapter 5
- This framework supports formal definitions of goals in temporal logics and already includes a formal notion of inconsistency. These are the concepts we need to reason about in Chapter 6

3.2 The KAOS Goal-Oriented Requirement Specification Language

This section introduces the KAOS goal-oriented specification language. It is illustrated on the e-signature case study. First of all, it briefly recalls the principles of conceptual modeling and graphical representations used in requirements engineering. It then presents the structure of the KAOS language and the components of this language. It also introduces consistency rules binding the components of the KAOS language.

Some fragments of the KAOS language are not used in this thesis. They are not presented here in order to keep the presentation simple; this is discussed in Section 3.2.9.

3.2.1 Conceptual Modeling

Requirements Engineering approaches generally involve three different levels of modeling: the meta-level, the domain-level, and the instance level.

Figure 3.1 illustrates these levels on the KAOS framework and the e-purse example

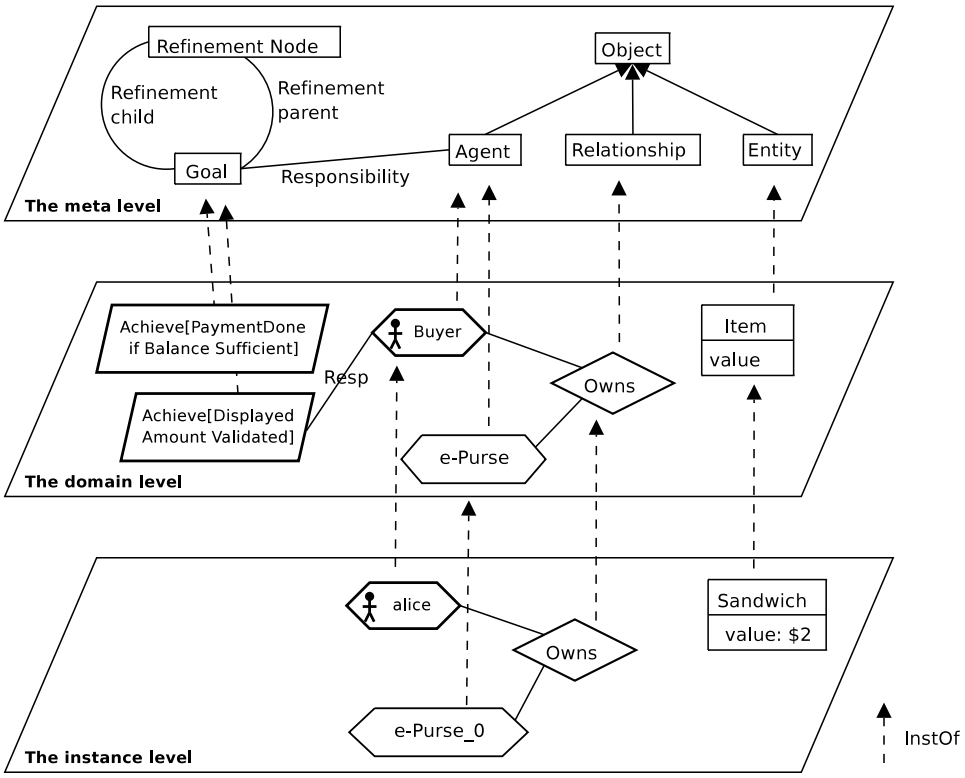

Figure 3.1: The meta, domain and instance levels

The *meta-model* provides domain-independent abstractions in terms of which domain-specific concepts are acquired. The meta-model is composed of meta-concepts (such as Goal, Agent, Relationship, etc.); meta-relationships (such as Refinement between Goals, Responsibility between Agent and Goal, etc.); meta-attributes of meta-concepts or meta-relationships (for example: Definition of Goal, Name of Entities, etc.); and *meta-constraints* on meta-concepts and meta-relationships (for example: "the vocabulary used in the definition of a goal must be defined in the object model").

The meta-model defines the language and the structure of requirements models. This language is defined so as to support a requirements elaboration method. For instance, the goal-oriented requirements elaboration method relies on the concepts of goal and refinement.

The *domain model* is composed of domain-specific instances of meta-concepts and meta-relationships. In Figure 3.1, the *Item* concept is an instance of the *Entity* meta-concept, the goal *Achieve[Payment done If Bal-*

ance Sufficient] is an instance of the Goal meta-concept, etc. The components of a requirements model are thus acquired as domain-specific instances of meta-concepts, linked by instances of meta-relationships, and characterized by instances of meta-attributes. These components must furthermore satisfy the meta-constraints of the meta-model. The *instance model* refers to specific instances of domain-level concepts. A specific *sandwich* is an instance of the domain-level concept of *Item*. Similarly, in Figure 3.1, Alice is an instance of Buyer.

3.2.2 Structuring the Framework

The KAOS language is a multiparadigm language. A KAOS model is composed of several sub-models related through concept sharing and *inter-model consistency rules* (Figure 3.2).

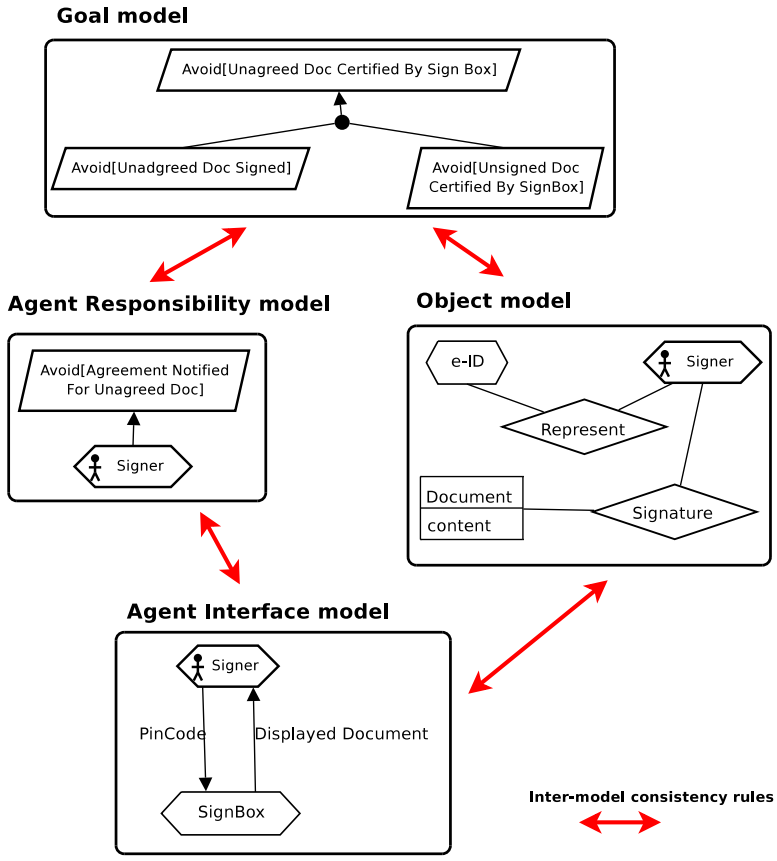

Figure 3.2: Structuring the KAOS meta-model

The *goal model* is the driving model of the KAOS language. It declares the goals of the system-to-be. A goal defines an objective the system-to-be should meet, usually through the cooperation of multiple agents. For instance, a goal of the e-signature system is *Avoid[Unagreed Doc Signed]*. It states that "*A document should be signed only if the signer agrees on the document*". Goal-refinement links relate a goal to a set of subgoals. A set of subgoals refines a parent goal if the satisfaction of all subgoals is sufficient for satisfying the parent goal [DvLF93, Dar95, DvL96]. As an example, the goal *Avoid[Unagreed Doc Certified By Sign Box]* is refined in Figure 3.2 into the subgoals *Avoid [Unagreed Doc Signed]* and *Avoid [Unsigned Doc Certified By SignBox]*.

The *object model* declares the objects of interest in the application domain. An object is classified as entity, relationship, event or agent dependent on whether the object is autonomous, subordinate, instantaneous or active. The object's characteristics are declared as attributes and relationship links to other objects.

There are *inter-model rules* between the object model and the goal model. On one side, the object model declares the vocabulary to be used in the definition of goals; on the other side, these definitions refer to concepts to be declared in the object model.

The *agent responsibility model* declares responsibility assignments of goals to agents. Agents include software components that exist or are to be developed, external devices, and humans in the environment. Responsibility assignment provides a criterion for stopping the goal refinement process. A goal assigned as the responsibility of a single agent is not refined any further. The meaning of a responsibility assignment is that the agent responsible for a goal is the only one required to restrict its behavior so as to ensure that goal [Fea87].

The *agent interface model* declares which objects are monitored and controlled by each agent. In the agent interface model of Figure 3.2, the Signer agent controls the PinCode encoded into the SignBox he is standing at and monitors the document displayed by it.

There is an *inter-model consistency rule* between the agent interface and the corresponding responsibility assignment of a goal to an agent. Roughly speaking, a goal can be assigned as the responsibility of an agent only if the goal is stated in terms of objects that are monitorable and controllable by the agent and if the agent can behave so that it enforces the goal. This is known as the realizability meta-constraint. It is presented in Section 3.2.8.3.

3.2.3 Graphical Representation

Concepts and associations from requirements models are commonly represented by graphical diagrams, like the one shown in Figure 3.3. The appearances of concepts and relationships in a diagram are inherited from the meta-concepts or meta-relationships they instantiate.

In the KAOS method, a diagram is only a partial view of the overall requirements model. Moreover, diagrams can include elements coming from different models of the framework. For instance, a diagram can represent elements coming from the object model together with goals.

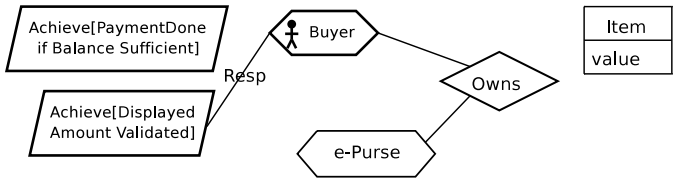

Figure 3.3: Diagrammatic representation of requirements models

3.2.4 Object Model

An *object* instance is a "thing" that can be distinctly identified. A domain-level object describes a set of such instances that share some common characteristics. A specific e-purse is an example of an object instance. The domain-level object e-purse is a conceptual abstraction characterizing all e-purse instances. The KAOS object meta-model distinguishes among different kinds of objects. An Object is an *Entity*, *Relationship* or *Agent*, depending on whether the object is autonomous, subordinate or active.

Each domain-level object has a name and a definition. The *name* of the object is used to identify the object. The *definition* of an object is a natural language statement that should provide a precise interpretation for what is one instance of this object and what is not so that one can tell whether or not a particular object instance is an instance of the domain-level object.

3.2.4.1 Entities

An *entity* is an object. All the instances of an entity share common features, that is: they have the same attributes (with different values) and participate to the same associations. They can be distinguished from one another; they have their own identity. Each entity is identified by a name and has a definition that is used to define which instances are a member of the domain-level entity. Entities can have attributes.

For instance, an electronic document in the e-signature system is an entity. The *document* entity has an attribute *content* that represents the text of the document. Documents are defined as follows:

Entity Document
  Definition A electronic document that can be signed, typically a contract
  Attribute *content: string* the text of the document

Entities are graphically represented by boxes with their attributes underneath, as in Figure 3.2, "Object model" part.

3.2.4.2 Agents

Agents are active objects such as humans, devices, or software components that play some role towards goal satisfaction. Some agents define the software-to-be whereas others define the environment. Our e-signature system includes Signer agents, Recipient agents, PC agents, e-ID agents, and SignBox agents. Agents have a local memory. We characterize this memory in Chapter 4.

Agents are declared by a name and a definition. The particular meta-features of agents are that they can monitor and control objects, and take responsibility for goals. Such features are described in the sections describing the agent interface and responsibility models.

For instance, the SignBox agent has an attribute keypad where the signer can dial her pin code. This agent is defined as follows:

Agent SignBox
Definition It is an electronic device connected to a PC that interfaces an e-ID to a signer. Unlike a classical PC that might be compromised by viruses, key loggers, root kits, and the like, the SignBox is a highly secure device.
Attribute *keypad: int* a standard keypad where signers can dial some PinCode

Agents are graphically represented by flattened hexagons, as in Figure 3.2 , "Object model" part. Human agents are sometimes graphically distinguished from software/hardware agents by annotating the hexagon representing human agent with a stylized drawing of human.

3.2.4.3 Associations

Associations are mathematical relations between objects and basic types. The former are called *members* and are defined through a *role*; the latter are attributes. Associations are declared by a name and a definition. The particular (meta-) features of associations are that they refer to entities or agent through role links. The role of an object in a relationship is the function that it performs in the relationship. It is defined by a name, a type, an optional multiplicity and a textual definition. This definition provides a precise interpretation of for what it means for an object instance to hold this role in the defined association. For instance, the *Represent* association is defined as follows:

Association Represent
Definition an e-id represents a signer if it mentions his name and allows to perform signature of electronic document with his e-signature.
Role *eid:e-ID* the e-id representing some signer
Role *signer(0:1):Signer* the signed represented by the e-id

The multiplicity of a role in an association is a pair (mincard, maxcard) that denotes the minimum and maximum number of instances of the relationship in which a given object instance can be involved simultaneously, respectively. Multiplicity constraints in the object model can only be used to state domain properties. For instance, the *Inserted* association is defined as follows:

Association Inserted
Definition represents the fact that an e-id is inserted into a SignBox
Role *e-id(0:1):e-ID* the e-id inserted into the SignBox
Role *signbox(0:1):SignBox* the SignBox the e-id is inserted in

The (0:1) multiplicity of role e-id specifies that a given *e-ID* can be inserted into at most one *SignBox* at a time. The (0:1) multiplicity of role SignBox specifies that at most one *e-ID* can be inserted into a given *SignBox* at a time.

This convention for multiplicity is borrowed from the OMT standard [BC95]. The UML standard specifies another convention for multiplicities, which is not considered in this thesis.

Associations serve not only capture notions from the physical world, but also abstract notions such as agreement of user about a document:

Association Agrees
Definition A signer agrees with a document iff she is willing to sign it
Role *who:Signer* the signer who is willing to sign something
Role *what:Document* the document that the signer is willing to sign

In this thesis, associations are graphically represented by a diamond as in Figure 3.2, "Object model" part.

3.2.4.4 Specifying Assertions over the Object Model

Entities and agents from the object model may be used as sorts in quantifications. Attributes define state variables; they are accessed through the standard "." selector. For example, the assertion

$$(\exists s\colon \text{Signer})(s.\text{name} = \text{'Alice'})$$

says that there must be at least one signer whose name is Alice.

Association classes define predicates over instance variables playing the

corresponding role. For example, the predicate

$$\text{Represent}(s,\ e)$$

is true in the current state if and only if the couple (s, e) belongs to the association *Represent*.

In this thesis, we sometimes use a more explicit notation for referencing associations within assertions. In this notation, attributes and members of the associations are explicitly named in the assertion referencing it. Our notation looks like this:

$$(\forall \text{e: e-ID})(\exists s\text{: Signer})\text{Represent}\langle\text{signer: } s,\ \text{eid: } e\rangle$$

The association *Represent* is defined in Section 3.2.4.3. The words *signer* and *eid* appearing in our notation refer to roles or attributes of the association. an association with roles *signer* and *eid* and no attributes. In this assertion, the notation Represent<signer: s, eid: e> denotes the underlying predicate Represent(s, e). This notation tolerates members to be referenced in any order, they do not need to have a particular position within the $<\ldots>$ brackets. Members that are not referenced are taken as existentially quantified at the nearest nesting level. For instance, the above assertion is equivalent to the following shorter one:

$$(\forall \text{e} : \text{e-ID})\text{Represent}\langle\text{eid: } e\rangle$$

In this assertion, the notation Represent<eid: e> denotes the underlying assertion:

$$(\exists s\text{: Signer})\text{Represent}(s, e)$$

This notation allows associations declared in a requirements model to be updated and given additional attributes or members without the need for rewriting all the assertions referencing it.

3.2.5 Goal Model

A *goal* is a prescriptive statement of intent about the considered system-to-be [vL01]. For example, *Achieve[Agreed Document Signed]* is a functional goal of the e-signature system.

Each goal has a name and a definition. The *name* of the goal is used to identify the goal. The *definition* of the goal is the statement represented by the goal expressed in natural language. Goals can also be defined formally with the adjunction of a *formal def* field. The language for defining goals formally is defined in the next section. For instance, the goal *Achieve[Agreed Document Signed]* is fully defined as follows:

Goal Achieve[Agreed Document Signed]
Definition Whenever a signer is willing to sign a document, this document is eventually signed with her signature.

The *pattern* of a goal is based on the temporal behavior required by the goal. The KAOS language distinguishes the following four goal patterns:

- **Achieve goals:** goals requiring that some property eventually holds
- **Cease goals:** goals requiring that some property eventually stops to hold
- **Maintain goals:** goals requiring that some property always holds
- **Avoid goals:** goals requiring that some property never holds

Goal patterns provide a lightweight way of declaring the temporal behavior of a goal without writing formal goal definitions. The pattern of a goal is often used as the first part of the goal name, like in the goal here above.

Security goals are a particular set of goals that require assets to be protected against non-authorized use. For example, the goal *Avoid [Unagreed Document Signed]* is a security goal of the e-signature system. It protects the signature asset against use by non-owners of the signature. It is defined as follows:

Goal Avoid [Unagreed Document Signed]
Definition If a signer has never been willing to sign a document, this document should not be signed with her signature

Unlike goals, *domain properties* are descriptive statements about the environment, for example, physical laws, organizational norms, etc.

3.2.5.1 Defining goals formally

Goals and domain properties can be formally defined in first order linear temporal logics [Koy92, Let01].

Formal assertions are interpreted over historical sequences of states. Each assertion is in general satisfied by some sequences and falsified by some other sequences. The notation

$$(H, i) \models P$$

is used to express that assertion P is satisfied by history H at time position i ($i \in T$), where T denotes a linear temporal structure assumed to be discrete. This linear structure is mapped to natural numbers starting at 1. We also write $H \models P$ as a shorthand for $(H, 1) \models P$.

States are global; the state of the composite system at some time position i is the aggregation of the local states of all its objects at that time position. The state of an individual object instance *ob* at some time position is defined as a mapping from *ob* to the set of values of all *ob*'s attributes and links at that time position. In the context of KAOS requirements, an historical sequence of states corresponds to a behaviour or scenario.

The KAOS framework imports the following classical operators for temporal referencing [MP92]:

$\circ$	*(in the next state)*	$\bullet$	*(in the previous state)*
$\Diamond$	*(some time in the future)*	$\blacklozenge$	*(some time in the past)*
$\Box$	*(always in the future)*	$\blacksquare$	*(always in the past)*
$\mathcal{U}$	*(always in the future until)*	$\mathcal{W}$	*(always in the future unless)*

The semantics of the above temporal operators is then defined as usual [MP92], for example,

$$
\begin{array}{rll}
(H,i) \models \circ P & \text{iff} & (H,i+1) \models P \\
(H,i) \models \Diamond P & \text{iff} & (H,j) \models P \text{ for some } j \geq i \\
(H,i) \models \Box P & \text{iff} & (H,j) \models P \text{ for all } j \geq i \\
(H,i) \models \bullet P & \text{iff} & i \geq 1 \text{ and } (H,i-1) \models P \\
(H,i) \models \blacklozenge P & \text{iff} & (H,j) \models P \text{ for some } j \leq i \\
(H,i) \models \blacksquare P & \text{iff} & (H,j) \models P \text{ for all } j \leq i \\
(H,i) \models P \,\mathcal{U}\, Q & \text{iff} & \text{there exists a } j \geq i \text{ such that } (H,j) \models Q \\
 & & \text{and for every } k, i \leq k < j, (H,k) \models P \\
(H,i) \models P \,\mathcal{W}\, Q & \text{iff} & (H,i) \models P \,\mathcal{U}\, Q \text{ or } (H,i) \models \Box P
\end{array}
$$

Note that $\Box P$ amounts to $P \,\mathcal{W}$ false. We will also use the standard logical connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\rightarrow$ (implies), $\leftrightarrow$ (equivalent), $\Rightarrow$ (entails), $\Leftrightarrow$ (congruent), @ (event) with

$$
\begin{array}{rll}
P \Rightarrow Q & \text{iff} & \Box(P \rightarrow Q) \\
P \Leftrightarrow Q & \text{iff} & \Box(P \leftrightarrow Q) \\
@P & \text{iff} & P \wedge \neg \bullet P
\end{array}
$$

There is thus an implicit outer $\Box$ operator in every entailment. We also use variable exponent a^+ and a^- to denote the next and previous value of state variable a.

In order to capture real-time properties, we introduce bounded versions of qualitative temporal operators in which the real-time bounds refer to the absolute time of the system, for example,

$\Diamond_{\leq T}$ *(some time in the future before time T)*
$\Box_{\leq T}$ *(always in the future up to time T)*

Within this thesis, we consider that time is discrete and that the amount of time between two successive states is 1 unit. The semantics of real-time operators is then defined as follows, for example,

$(H, i) \models \Diamond_{\leq T} P$ iff $(H, j) \models P$ for some j such that $i \leq j \leq i + T$
$(H, i) \models \Box_{\leq T} P$ iff $(H, j) \models P$ for all j such that $i \leq j \leq i + T$

For example, the goals *Achieve[Agreed Document Signed]* and *Avoid [Unagreed Document Signed]* are fully defined as follows:

Goal Achieve[Agreed Document Signed]
Definition Whenever a signer is willing to sign a document, this document is eventually signed with her signature.
Formal Def($\forall$ s: Signer, d: Document)
Agrees<who: s, what: d> $\Rightarrow$ $\Diamond$Signed<signer: s, document: d>

Goal Avoid [Unagreed Document Signed]
Definition If a signer has never been willing to sign a document, this document should not be signed with her signature
Formal Def($\forall$ s: Signer, d: Document)
Signed<signer: s, document: d> $\Rightarrow$ $\blacklozenge$Agrees<who: s, what: d>

This language for defining goals formally does not allow us to define confidentiality requirements. Defining confidentiality goals requires an extended logics [BDR04, BC93]. This is because confidentiality constraints the knowledge of agents and the knowledge of agents grows through observation and through deduction or inferences that agents can make. Deontic logics have been proposed for modeling confidentiality policies [BC93]. Here, we want to reason about confidentiality requirements. Chapter 4 presents an extended framework for defining confidentiality requirements.

3.2.5.2 Refinements and goal graphs

AND-refinement links relate a goal to a set of sub goals (called refinement); this means that satisfying all sub goals in the refinement is a sufficient condition in the domain for satisfying the goal. A goal can be AND-refined in several alternative ways. In this case, these are considered as alternative refinements and one of these alternatives must be chosen.

Formally, a set of goals $\{G_1, \ldots, G_n\}$ AND-refines (or refines for short) a goal G in a domain theory *Dom* if the following conditions hold [Dar95]:

1. $G_1, \ldots, G_n, \text{Dom} \models G$ (*completeness*)
2. $\wedge_{j \neq i} G_j, \text{Dom} \not\models G$ for each $i \in [1..n]$ (*minimality*)
3. $G_1, \ldots, G_n, \text{Dom} \not\models \text{false}$ (*consistency*)

The first condition requires that the satisfaction of the sub goals together with the satisfaction of domain properties in *Dom* is sufficient for satisfying the parent goal. The second condition requires that if a sub goal is left out of the refinement, the remaining sub goals are not sufficient for satisfying the parent goal. The third condition requires that the conjunction of the sub goals is logically consistent with the domain theory.

The formal definition of goals allows one to verify formally the completeness, minimality and consistency of goal refinements.

Figure 3.4 shows a portion of the goal graph for our e-signature system. The root goal there states that the signature box should formally certify that any electronic signature guarantees the signer's agreement of a certified document. The leaf goal on the left is an assumption under responsibility of the *Signer* agent whereas the leaf goal on the right is a requirement assigned to the *PC* software. Figure 3.5 shows another portion of goal graph for our e-signature system. It refines a functional goal of the system. These goals are considered as functional goals, meaning that we do not care about denial of service attacks in this case study. If we were considering these as security goals, appropriate techniques should be used in order to derive the requirements ensuring that the system would be robust against denial of service attacks.

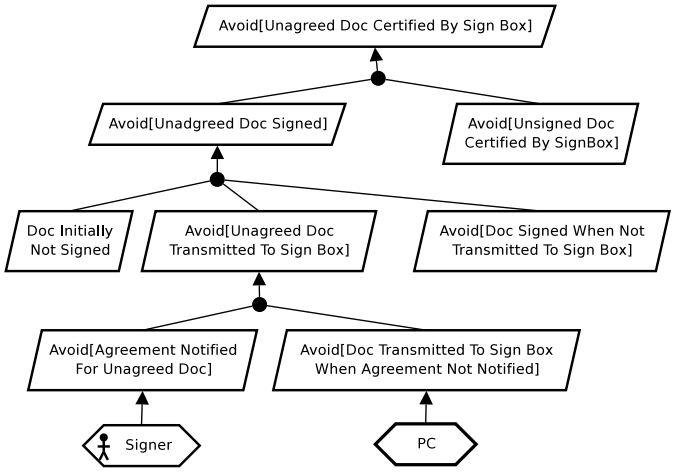

Figure 3.4: Goal graph refining a security goal of the e-signature system

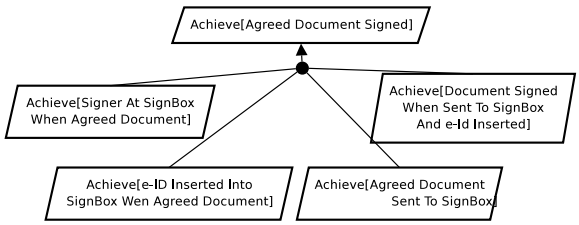

Figure 3.5: Goal graph refining a functional goal of the e-signature system

3.2.5.3 Modeling Conflicts

This section introduces the concepts of inconsistencies between goals. A divergence between goals corresponds to situations where some particular combination of circumstances can be found that makes the goals logically inconsistent. Such a particular combination of circumstances is called a *boundary condition* [vLDL98].

Let *Dom* denote a set of domain descriptions that capture the knowledge available about the domain [JZ93].

Definition (*divergence*): There is a divergence between assertions $A_1, \ldots A_n$ within a domain *Dom* **iff** there exists a boundary condition B such that the following conditions hold:

1. $\{Dom, B, \wedge_{i\in[1,\ldots n]} A_i\} \vdash$ false (*logical inconsistency*)
2. for every i: $\{Dom, B, \wedge_{j\neq i} A_j\} \nvdash$ false (*minimality*)
3. there exists a trace S and time position i such that $(S, i) \models B$ (*feasibility*)

The notation "$A \vdash B$" denotes that assertion B can, be logically derived from assertion A (or from set of assertions A, which is taken as the conjunction of the assertions in A). A *trace* is defined as a historical sequence of states, that is: a finite prefix of a history. The boundary condition captures a particular combination of circumstances which makes assertions $A_1, \ldots, A_n$ inconsistent if conjoined to it (see conditions (1) and (2)).

Condition (3) states that the boundary condition is satisfiable through one behavior at least, that is, there should be at least one scenario that establishes the boundary condition. Clearly, it makes no sense to reason about boundary conditions that cannot occur through some feasible agent behavior.

Definition (*conflict*): A conflict is a particular case of divergence in which $B =$ true

Divergences and conflicts are graphically represented by red diamond

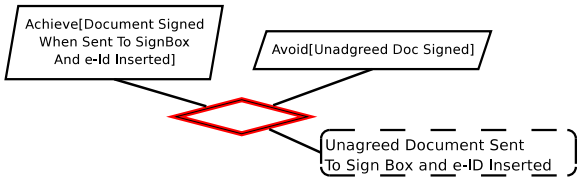

Figure 3.6: Divergence in the e-Signature System

with a link to the involved goals. The boundary condition of each divergence is informally summarized in an optional rounded box linked to the diamond. For instance, in the e-signature system, the functional goal *Achieve[Document Signed When Sent To SignBox and e-ID Inserted]* and the security goal *Avoid[Unagreed Doc Signed]* are divergent.

The boundary condition for this divergence is the case where a document is sent to a SignBox and is not agreed by the user represented by the e-ID inserted into the SignBox. In this case, the document should be both signed and not signed.

Divergences must be resolved. In this case, the functional goal must be weakened into *Achieve[Agreed Document Signed When Sent To SignBox and e-ID Inserted]*.

The definitions presented here need to be made further precise in order to provide automated tool support for detecting divergences. This discussion belongs to the contributions of this thesis; it is presented in Section 6.4.3.

3.2.5.4 Modeling Obstructions

First-sketch specifications of goals, requirements and assumptions tend to be too ideal; they are likely to be occasionally violated in the running system due to unexpected agent behavior [vLDM95, vLL00]. The objective of obstacle analysis is to anticipate exceptional behaviors in order to derive more complete and realistic goals, requirements and assumptions.

These failures are modeled in requirements models through the concept of obstacle. *Obstacles* are a dual notion to goals; while goals capture desired conditions, obstacles capture undesirable (but nevertheless possible) ones. An obstacle obstructs some goal, that is, when the obstacle gets true the goal may not be achieved. The term "obstacle" denotes a goal-oriented abstraction, at the requirements engineering level, of various notions that have been studied extensively in specific areas, such as hazards that may obstruct safety goals [Lev95] or threats that may obstruct security goals [Amo94, vL04], or in later phases of the software lifecycle, such as faults that may prevent a program from achieving its specification [CL95, Gar99a].

Definition (*obstacle*): Let G be a goal and Dom a set of domain properties. An assertion O is said to be an obstacle to G in Dom iff the following two conditions hold:
1 $O, Dom \models \neg G$ *(obstruction)*
2 $O, Dom \not\vdash$ false *(domain-consistency)*

For instance, there is an obstacle to the goal *Achieve[e-id inserted into SignBox when willing to Sign]* in that a chewing gum blocked in the card reader may make it impossible to insert an e-ID into the card reader.

3.2.6 Agent Responsibility Model

A *requirement* is a goal under responsibility of a single agent of the software-to-be. An *assumption* is a goal under responsibility of a single agent of the environment. The semantics of responsibility assignment of a goal to an agent is that the agent must restrict its behavior so as to ensure the goal [Fea87].

Responsibility links declare that instances of an agent class are responsible for the corresponding leaf goal. Instance declarations, attached to such links, specify which agent instance is responsible for the corresponding leaf goal instantiated to specific object instances [Let01]. For example, consider the leaf goal on the left in Figure 3.4:

Assumption Avoid [Agreement Notified For Unagreed Doc]
Definition The PC connected to the signature box where an e-ID is inserted should not be notified that the e-ID owner agrees with a document if she does not.
Formal Def($\forall doc$: Document, sgn: Signer, p: PC, sb: SgnBox, e : e-ID)
$\neg$Agrees<what: doc, who: sgn>$\wedge$ represent<signer: sgn, eid: e>
$\wedge$Inserted<eid: e, signbox: sb>$\wedge$ Connected<signbox: sb, pc: p>
$\Rightarrow \neg$AgreementNotified<signer: sgn, document: doc, pc : p)

The responsibility link attached to this assumption is made further precise as follows:

Responsibility
Under Responsibilty Of Signer
Responsible Of Avoid[Agreement Notified For Unagreed Doc]
Definition The signer at the PC is the one responsible for not notifying the PC of untrue document agreement
Formal Def($\forall sgn$: $Signer$, p: PC, sb: $SgnBox$, $e : e - ID$)
At<user: sgn, pc: p>$\wedge$ Inserted<eid: e, signbox: sb>$\wedge$
Connected<signbox: sb, pc: p>
$\Rightarrow ResponsibleFor(sgn,$ Avoid[Agreement Notified
For Unagreed Doc]$(doc, sgn, p))$

In the above responsibility instance declaration, the assertion

$$\text{Avoid[Agreement Notified For Unagreed Doc]}(doc, sgn, p)$$

states that *it should not be notified to PC p that signer sgn agrees on document doc if she does not*. The goal is thus instantiated to specific instance variables. Goals actually represent goal class. An instance-level goal is a goal class where the outer universal quantifiers are resolved to a specific instance. It is formally referenced through the parameter notation presented here.

3.2.7 Agent Interface Model

Agent interfaces are declared through Monitoring and Control links between agents and objects' attributes.

The meaning of a *Monitoring* link between an agent and an object attribute is that the agent directly monitors ("reads") the value of the attribute.

As an example, the signer standing at a certain SignBox monitors what the SignBox displays on its screen. Similarly, the SignBox monitors the value entered in the keypad attached to it.

The meaning of a *Control* link between an agent and an object attribute is that the agent directly controls ("writes") the value of the attribute. In other words, an agent controls an attribute if it is capable of controlling state transitions for that attribute. We also consider that attributes controlled by an agent are observable by that agent as well.

As an example, the SignBox controls what is displayed on its screen. Similarly, the Signer standing at a certain SignBox controls the value entered in the keypad attached to it.

3.2.8 Consistency Rules

This section presents the consistency rules binding the sub-models that compose the KAOS language.

3.2.8.1 Consistency between the object and goal models

There is a consistency rule relating the goal and object models:

Every vocabulary element used in the definition of goals must be declared in the object model.

Consider for instance the following goal:

Goal Achieve[Agreed Document Signed]
Definition Whenever a signer is willing to sign a document, this document is eventually signed with her signature.
Formal Def($\forall$ s: Signer, d: Document)
Agrees<who: s, what: d> $\Rightarrow$ $\Diamond$Signed<signer: s, document: d>

The formal definition references the concepts Signer, Document, Agreement, and Signature. In order to satisfy the meta-constraint between the goal and object models, these concepts mut be declared in the object model, as done in Section 3.2.4. This meta-constraint can also be used to derive concepts of the object model from the delclaration of goals.

3.2.8.2 Consistency between the object and the agent interface models

The *Control* meta-relationship must satisfy the *unique control* meta-constraint requiring that in each alternative an object's attribute is controlled by at most one agent. At the outer-level of the language, the unique control constraint is defined as follows:

$$\text{if Control}(ag, Ob.Attr) \text{ then}$$
$$\text{there is no } ag' \neq ag \text{ such that Control}(ag', Ob.Attr)$$

The *unique control* constraint requires that the predicate R involved in an instance declaration of the form

$$R(ag, o) \Rightarrow Ctrl(ag, o.Attr)$$

satisfies the following condition:

$$R(ag, o) \Rightarrow \neg(\exists ag')ag' \neq ag \wedge R(ag', o)$$

The unique control constraint is used to avoid interference problems between concurrent executions of agents. The unique control constraint is an important and necessary feature of the language. It is necessary to be able to assign goals as the responsibility of single agents.

3.2.8.3 Realizability

Realizability is a condition for ensuring that a set of requirements can be enforced together by an agent responsible for them given its interface (what it controls and what it monitors). Basically, a set of requirements

is realizable by an agent if there exists a transition system for the agent whose inputs and outputs are the variables monitored and controlled by the agent, respectively such that this transition system enforces the requirements [PR89a, PR89b, Let01].

We can distinguish three variations around this concept of realizability:

- Agents can be modeled either as being able to react instantaneously to a stimulus (*synchronous agents*) or require at least one time unit to react to a stimulus (*asynchronous agents*)[1]. Synchronous agents are also referenced to as the synchrony assumption. The *synchrony assumption* states that software agents can react instantaneously, that is: if there is a stimulus at time t, they can produce the adequate reaction at time t [HJL96, BG89]. The rationale behind this unrealistic assumption is that software agents are infinitely faster, compared to agents in their environment.

- Agents might have to provide suitable outputs either for all possible inputs or only to inputs satisfying some domain properties [JZ93, Sch02a]. This is referred to as *domain-dependent realizability* and *domain-independent realizability*.

- A requirements-engineering specific variation imposes the transition system not to be more restrictive than the considered requirements [LvL02]. This is referred to as *strong realizability*. The other alternative is referred to as the *weak realizability*. The purpose of strong realizability is to ensure that (1) the variables appearing in the formal definition of a requirement belong to the vocabulary of the agent responsible for it, (2) that enforcing the goals does not require to predict the future value of some monitored variables, and (3) that the agent responsible for the requireement controls enough variables of the requirments to enforce it. These three conditions, in conjunction with weak realizability have been shown to be equivalent to the strong realizability condition [LvL02, Let01].

In the KAOS framework, we are willing to derive asynchronous agents because they are more realistic than synchronous ones [LvL02]. The realizability used in the KAOS framework is strong and domain-independent [LvL02].

In this thesis however, we focus on weak realizability. The purpose of strong realizability is related to vocabulary concerns, in addition to weak realizability [LvL02]. We therefore use the weak asynchronous domain-independent realizability, that will be referred to as "the" *realizability*

[1] These terms might have a different meaning in other temporal logic -based frameworks such as LTSA [LKMU05, MK99]

throughout this thesis. We use the notation Realizable(ϕ, ag) to denote that assertion ϕ is realizable by agent *ag* defined by its interface.

The weak realizability property is equivalent to the existence of a winning strategy for the agent-to-be-implemented with respect to its requirement [PR89a, PR89b, Mor94]. This strategy might be infinite in time, hence it is often defined as a fixpoint formula on a Büchi automaton that is equivalent to the considered temporal logic formula. Here, we prefer another, more explicit definition.

The realizability condition for a temporal specification ϕ by a synchronous agent is defined as follows:

$$(\forall m_1)(\exists c_1)(\forall m_2)(\exists c_2)\ldots(\forall m_n)(\exists c_n)\ldots H \models \phi$$

where H represents any history such that at any time position t, the value of the controlled and monitored variables is given by m_t and c_t, respectively.

The intuition behind this formula is that at any time t, whatever is the value for the monitored variables decided by the environment, m_t, the agent is able to provide an output ct that is acceptable with respect to its requirements ϕ. The output of the agent c_t can be decided by the agent based on all the past history and on the input of the environment at the same time m_t.

The realizability condition in the case of an asynchronous agent is defined as follows:

$$(\exists c_1)(\forall m_1)(\exists c_2)(\forall m_2)\ldots(\exists c_n)(\forall m_n)\ldots H \models \phi$$

The difference with the previous one is that at time t, the output of the agent c_t can be decided based on all the past history but not on the input of the environment at the same time m_t.

Synchronous and asynchronous realizability are related by the following property:

Property 1 *An assertion is not realizable by an asynchronous (resp. synchronous) agent iff its negation is realizable by the synchronous (resp. asynchronous) environment. The latter controls the variables monitored by the agent and monitors the variables controlled by the agent. This holds if the vocabulary of the specification is included in the set of variables controlled or monitored by the agent.*

This property trivially follows from the equivalence between $\neg\exists x : P(x)$ and $\forall x : \neg P(x)$.

Realizability and satisfiability are related to each other via the following properties:

Property 2 *A temporal assertion is satisfiable iff it is realizable by an agent that controls every variable referenced in this assertion.*

Satisfiability of temporal assertion is defined as $\exists H : H \models \phi$, this trivially proves the property.

Property 3 *If an assertion is realizable by an agent, she can still realize it if she gets to control more variables, possibly ones that she monitored in the first place.*

This arises because $(\forall x)(\exists y)P(x, y)$ implies $(\exists x, y)P(x, y)$.

Property 4 *In a domain-independent context, if an agent monitors or controls all the variables from the vocabulary of an assertion, then she can realize it iff she can realize it if she is given the capability to control additional variables not in her vocabulary.*

This arises because $(\exists x)P(x)$ implies $(\exists x, y)P(x)$.

Property 5 *Weak realizability of assertions by a given agent is monotonic with respect to the subset relationship taken as order on sets of assertions:*

$$(\forall\ ag : \mathit{Agent},\ D, D' : \mathit{set\ of\ assertions})$$
$$\neg \mathit{Realizable}(D, ag) \rightarrow \neg \mathit{Realizable}(D \cup D', ag)$$

This arises because $\neg(\exists x)P(x)$ implies $\neg(\exists x)P(x) \wedge Q(x)$.

3.2.9 Overview of the full framework

The KAOS framework features additional models that have not been presented here. We only presented the subset of interest in the context of this thesis. An exhaustive description of this framework can be found in [vL03a]. An exhaustive description of the requirements elaboration process can be found in [Let01].

The *operation model* is a fifth model that is not presented here. Basically, it is a reformulation of agent responsibility model where goals are regrouped into operations. An agent can perform an operation. An operation operationnalizes requirements under the responsibility of the agent. A set of operation is semantically equivalent to a transition system on an object model. The operation model comes along with consistency rules binding it to the object, agent and responsibility models.

3.3 Goal-Oriented Requirements Elaboration Method

This section presents the goal-oriented requirements model elaboration method. It is illustrated on the e-purse system.

Figure 3.7 depicts a simplified process for elaborating requirements model in a goal-oriented way. It basically consists in deriving the requirements for a future system from high-level goals. The steps are ordered by data dependencies and generally intertwined. They are detailed in the next subsections.

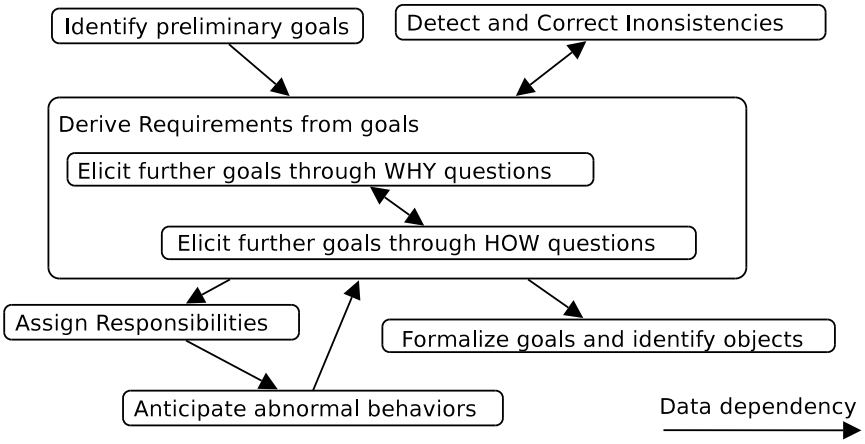

Figure 3.7: Goal-oriented elaboration of requirements method

3.3.1 Identifying Preliminary Goals

The first step of the elaboration method consists in identifying preliminary goals from various sources: interviews, analysis of available documentation to find out problematic issues with the existing system, objectives that are explicitly stated about the envisioned one, usage scenarios, operational choices whose rationale has to be elicited, etc. Some preliminary goals identified from the problem statement of e-purse system are given below.

Goal Avoid[Payment Done If Amount Not Validated]

Definition No payment should be done from an e-purse to a terminal, when the card is inserted in the terminal, the amount is entered by the seller in the terminal if the OK button on the buyer side of the terminal is not pressed.

Goal Achieve[Correct Item Value Entered In Terminal]
Definition The value of an item that is sold by a seller and bought by a buyer should be entered in the seller's terminal when the buyer wants to buy it.

Goal Maintain[e-purse balance positive]
Definition The balance of an e-purse should always be positive or null

Goal Avoid[Buyer Attacked For e-purse]
Definition Buyers should not be physically attacked for their e-purse

Goal Maintain[Buyer Wealth Unchanged When Paying]
Definition The sum of the money stored on the e-purse of a buyer and the value of the good he owns should not be changed when the buyer buys a new item and pays for it with her e-purse

3.3.2 Identifying Objects from Preliminary Goals

The definition of goals drives the identification of objects and attributes to be declared in the object model. Through application of the consistency rule between the goal model and object model, the vocabulary used in the formal definition of goals gives rise to the declaration of objects and attributes in the object model.

We can therefore define the concepts of e-purse, buyer, item and terminal from the goals defined here above. These concepts are fully defined as follows:

Agent Buyer
Definition human agent that might wish some items and buy them

Agent Seller
Definition human agent that sells items to the buyers

Agent ePurse
Definition smart card that contains e-money. It is held by a buyer.
Attribute *balance: int* The amount of e-money stored in the ePurse

E-purses are modeled as agents because they can be assigned requirements, as they have an active processing capability.

Agent Terminal
Definition Terminals are small terminal in which ePurses can be inserted by buyers. It stores e-money collected from all the payment processed to it.
Attribute *balance: int* The amount of e-money presently stored in the terminal
Attribute *enteredamount : int* The amount entered in the terminal when a buyer is buying an item

Entity Item
Definition something that might be bought by a seller to a buyer.
Attribute *value: int* value of items do not change in the time and is known to every buyer and seller.

Association Attacked
Definition This denotes the fact that the buyer is physically attacked for her e-purse
Role *buyer:Buyer* The buyer under attack

3.3.3 Eliciting new goals Through HOW and WHY questions

Asking WHY questions enables one to identify higher-level goals that provide rationale for the initial goals. Identifying higher-level goals is important for understanding the initial goals, and for identifying other important subgoals that may have been overlooked in the first place.

There are no clear-cut criteria for knowing when to stop asking WHY questions [ZJ97]. The quest for higher-level goals should remain within the system's subject matter.

Asking WHY question about the goal *Achieve[Item Value Entered In Terminal]* yields the goal *Achieve[Wanted Item Paid]*, which is a functional goal defined as follows:

Goal Achieve[Wanted Item Paid]
Definition When a buyer wants an item sold by a seller, there must be a payment from her e-purse to the seller's terminal

This goal requires to be reworked because it is conflicting with the goal stating that the balance must not be negative and the goal stating that the balance of her e-purse should be decreasd by the value of the purchased item. This is the purpose of the step devoted to detecting and resolving inconsistencies. Put in other words, a buyer might not be able to pay for everything she wants. We will present some of these conflict in the next section.

Another step of the requirements elaboration process consists in refining goals until reaching subgoals that can be assigned to individual agents. Subgoals are identified by asking HOW questions about the goals already identified.

Through a HOW question, we can identify the goals refining *Achieve[Wanted Item Paid]*. These are the already identified goal *Achieve[Item Value Entered In Terminal]*, and the two following new goals:

Goal Achieve[ePurse Inserted in Terminal When Wanted Item]
Definition When a buyer wants an item sold by a seller, she should insert her e-purse into the buyer's terminal and not remove it before payment is done

Goal Achieve[Payment Done When e-purse inserted and value entered]
Definition When an e-purse is inserted into a terminal and an amount has been entered into the terminal, a payment should take place from the e-purse to the terminal

This process yields the refinement shown in Figure 3.8.

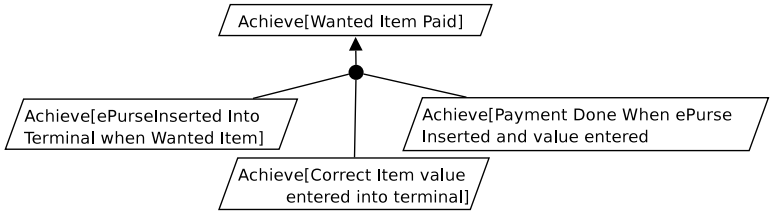

Figure 3.8: Refining the goal *Achieve[Wanted Item Paid]*

The elaboration of the goal graph can be carried on, for instance by exploring the graph upwards. We question why the functional goal *Achieve[Wanted Item Paid]* must be considered. This goal actually refines the functional goal *Achieve[Wanted Item Delivered]* defined as follows:

Goal Achieve[Wanted Item Delivered]
Definition When a buyer wants an item sold by a seller, the item should eventually be delivered to her

Again, we can identify an additional goal by completing the refinement of this new goal. The following goal is found by questioning how the goal *Achieve[Wanted Item Delivered]* is enforced:

Goal Achieve[Paid Item Delivered]
Definition When a payment has been done from an e-purse to a terminal in order to pay for an item wanted by the owner of the ePurse and sold by the owner of the terminal, the item should be delivered to the owner of the epurse

This goal also contributes to the already identified security goal *Maintain[Buyer Wealth Unchanged When Paying]*. The goal *Achieve[Paid Item delivered]* is therefore both a security goal and a functional goal.

Some refinements are inherently incomplete, especially when the goal cannot be formally defined, as for *Avoid[Buyer Attacked For e-purse]*. We therefore provide incomplete refinement. For instance, in order to avoid the owner of an e-purse to be attacked for her money, we must ensure that the balance of her e-purse is kept confidential. This gives rise to the following goal:

Goal Maintain[e-PurseBalanceConfidential]
Definition E-purse balances shall be known only to the e-purse holder and the e-purse itself

The presented language does not enable us to define this goal formally. It is extended in Chapter 4 with the necessary expressiveness for formally defining confidentiality goal.

3.3.4 Detecting and Correcting Conflicts

Conflict analysis can be carried in parallel with the goal model elaboration. Whenever a conflict is detected, the goal graph elaboration should be suspended and the conflict must be resolved because some conflicting goals might be modified by the conflict resolution. It is useless to spend time refining a goal that must be corrected because the refinement might need reworking as well.

For instance, two goals identified in this section are divergent. These are the goals *Avoid[Payment Done If Amount Not Validated]* and *Achieve[Payment Done When e-purse inserted and value entered]*. The boundary condition states that an e-purse is inserted into a terminal and an amount is encoded into the terminal, however, the OK button of the terminal is never pressed. A diagrammatic notation summarizes this divergence in Figure 3.9.

The resolution here consists in weakening the functional goal *Achieve[Payment Done When e-purse inserted and value entered]* into the following one:

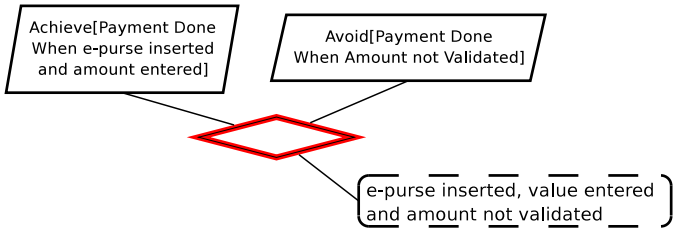

Figure 3.9: Divergent goals from the e-purse system

Goal Achieve[Payment Done **If Amount Validated**]
Definition A payment shall be done for some amount, from an e-purse to a terminal, when the card is inserted in the terminal, the amount is entered by the seller in the terminal, **and the OK button of the terminal is pressed**.

The weakened goal cannot replace the goal *Achieve[Payment Done When e-purse inserted and value entered]*in the refinement presented in Figure 3.8 because nothing prescribes that the OK button must be pressed in the refining goals. An alternative refinement must be proposed, which includes an additional goal prescribing that the of button must be pressed if the entered is equal to the value of the wanted item. This alternative refinement is presented in Figure 3.10.

Goal Achieve[Entered Amount Validated if Accurate]
Definition The OK button must be pressed if the entered amount it is equal to the value of the wanted item

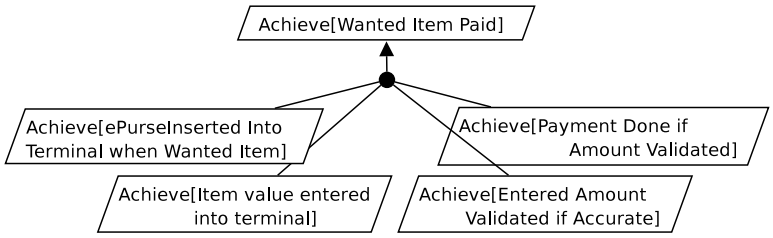

Figure 3.10: Alternative refinement for goal *Achieve[Wanted Item Paid]*

3.3.5 Assigning Responsibilities

Once fine-grained goals can be enforced by a single agent, given its interface, we can assign the responsibility of this goal to this agent. Agent interface can also be derived from goals assigned to them [Let01].

For instance, the goal *Achieve[ePurse Inserted in Terminal When Wanted Item]* can be enforced by the Buyer agent because a buyer monitors her will to buy an item and controls the insertion of her e-purse into the Seller's terminal. This responsibility assignment is shown in Figure 3.11.

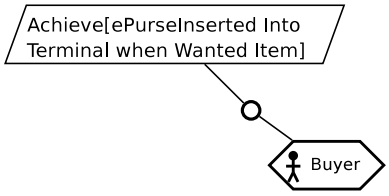

Figure 3.11: Responsibility assignment in the e-purse system

3.3.6 Anticipating Abnormal Behaviors through Obstacle Analysis

Obstacle analysis may take place whenever responsibility assignments are performed. Obstructions are detected first, and resolution to the identified obstacles can be proposed. Resolutions might deliver additional requirements that make the system robust against the condition described by the obstacle.

For instance, an obstruction in the ePurse case study is shown in Figure 3.12. The domain theory of ePurses includes the domain property that a chewing gum can prevent an ePurse from being inserted in a terminal when blocked in its card reader. Therefore, the obstacle *Chew Blocked In Terminal Card Reader* obstructs the assumption *Achieve[ePurse Inserted in Terminal When Wanted Item]*.

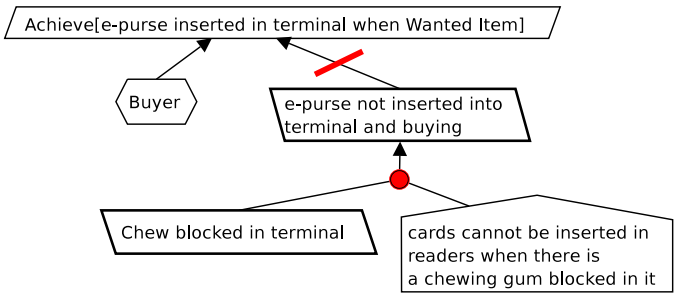

Figure 3.12: Obstacle analysis on an assumption

Obstacles can be refined like goals by means of regression techniques as exposed in [vLL00]. This is not discussed in this thesis.

The generated obstacles need to be resolved in some way or another. Different strategies may be applied in order to resolve an obstacle. The

strategies can be classified into three broad classes dependent on whether the obstacle is eliminated, reduced, or tolerated. Figure 3.13 presents the different strategies along with some examples of resolution for the obstacle of Figure 3.12.

Obstacle resolutions can be inserted into goal diagrams. A graphical example of obstacle resolution is shown in Figure 3.14. This resolution is an application of the *obstacle mitigation* resolution strategy.

Goals introduced in order to resolve obstacles are also subject to the refinement process and can give rise to new requirements / assumptions.

3.4 Summary

The work reported in the thesis is based on an existing goal-oriented requirement elaboration method, called KAOS. This method consists in identifying goals and refining them into subgoals until the latter can be assigned as responsibilities of single agents. The method supports the exploration of alternative goal refinements, and alternative responsibility assignments of goals to agents, resulting in alternative system proposals in which the boundary between the automated system and its environment may be quite different.

This chapter has described a fragment of the KAOS goal-oriented language and methods. It also pointed out the need for extensions of the language in order to model confidentiality requirements. This chapter also introduced the concept of divergence that might appear in goal models. This thesis proposes a technique for detecting them.

- **Goal substitution:** Identify an alternative goal refinement for some higher-level goal, in which the obstructed goal and obstructing obstacle are no longer present.

  Example: Use RFiD technology instead of contact-based communication, so that there is no need to insert the e-purse into the terminal anymore

- **Agent substitution:** Consider alternative agent assignments so that the obstacle can no longer occur.

- **Obstacle prevention:** Add a new goal requiring that the obstacle be avoided.

  Example: Design card reader so that chewing gums cannot be blocked in it.

- **Goal Deidealization** Transform (weaken) the goal so that it covers the behaviors captured by the obstacle

  Example: weaken the goal into: *Achieve[e-purse Inserted In ... When ...* ***Except If Blocked Chew****]*

- **Domain transformation** Transform the domain within which the software-to-be operates so as to make the obstruction disappear.

  Example: Change the recipe of chewing gums so that they cannot block card readers anymore

- **Obstacle reduction** Reduce the occurrences of the obstacle

  Example: Beep whenever somebody tries to block a chewing gum in the terminal

- **Goal restoration** Add a new goal stating that if the obstacle condition becomes true then the obstructed goal assertion should be satisfied again in some reasonably near future.

- **Obstacle mitigation** Add a new goal to attenuate the effects of obstacle occurrences

  Example: add the goal *Achieve[Stuck Chew Removed From Card Reader]*

- **Do-nothing** Tolerate obstacle occurrences without any resolution action

Figure 3.13: Strategies for resolving obstacles

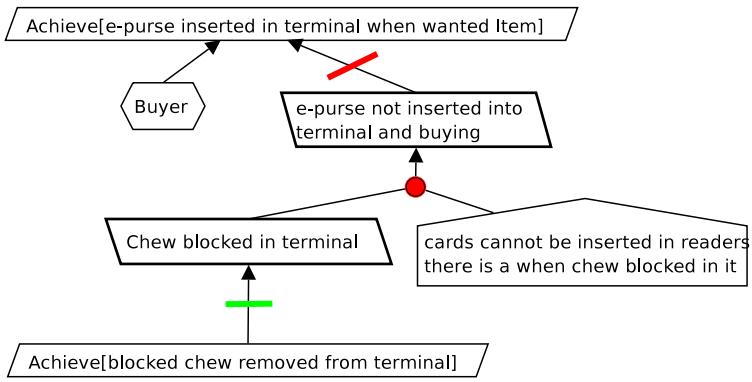

Figure 3.14: Resolving the obstacle *Chew Blocked In Terminal*

Chapter 4

Reasoning about Confidentiality at Requirements Engineering Time

This chapter describes an extended framework for formally specifying and reasoning about confidentiality goals at requirements engineering time [LvL05]. It is based on knowledge logics [FHMV95]. Our extension makes it possible to automate checks of requirements models against confidentiality claims and discover confidentiality violations at requirements engineering time. Counter-measures must then be found to yield new security requirements, possibly confidentiality requirements.

The chapter is organized as follows: Section 4.1 further motivates this contribution on the basis of an example from the e-purse case study; Section 4.2 introduces some background on knowledge logics; Section 4.3 extends an existing specification framework with a formal model of knowledge of unauthorized agents; Section 4.4 presents a taxonomy of confidentiality requirements patterns that relies on those grounds; Section 4.5 presents a proof-of-concept tool for early checking of requirements models against confidentiality claims; Section 4.6 presents how such tool can be applied to iteratively identify the security requirements enforcing a confidentiality goal; Section 4.7 discusses some related work.

4.1 Motivation: Analyzing Confidentiality Requirements

At requirements engineering time, we should ensure that confidentiality goals are enforced through adequate security requirements, possibly confidentiality requirements. For instance, the e-purse system should be designed so that the balance of an e-purse is not disclosed to unauthorized agents. Similarly, the e-signature system should behave so that the password of a signer is not disclosed to unauthorized agents.

Information disclosure can be direct or indirect. In the former case, confidential information is directly disclosed to some unauthorized agent. In the latter case, the agent gets to know some non-protected information and acquires forbidden knowledge by performing deductive reasoning on such information. This requires that we use extended logics to capture how agents' knowledge evolves with respect to the values he can know. We also need to make its memory and reasoning capabilities precise (see Section 3.3.3) [BDR04, LvL05].

In order to automatically detect confidentiality violations at requirements engineering time, we need to formalize confidentiality requirements such as the following:

Goal Maintain [e-Purse Balance Confidential]
Definition E-purse balances shall be known only to the e-purse holder and to the e-purse itself

Remember that e-purses are modeled as agents in Section 3.3.2 because they are assigned responsibilities.

This confidentiality goal was introduced in Section 3.3.3 in the refinement of *Avoid[Buyer Attacked for e-purse]*. The idea is that potential attackers are much more likely to be tempted by attacking the e-purse owner if they have some knowledge about the balance of the owner's e-purse.

To formalize confidentiality requirements, we need epistemic constructs that capture what agents may or may not know in specific states [BDR04]. The epistemic operator Knows_{ag} was introduced in [vLL00] to capture accuracy requirements and inaccuracy obstacles. It is defined as follows:

$$\text{Knows}_{ag}(P) \equiv \text{Belief}_{ag}(P) \wedge P$$

The operational semantics of the epistemic operator $\text{Belief}_{ag}(P)$ is: *"P is among the properties currently stored in the local memory of agent ag"*. An agent thus knows a property if that property is found in its local memory and it is indeed the case that the property holds. This chapter will define how properties enter and leave this memory in the context of security.

The following particularization of this construct was used in [vL04] to capture malicious violations of confidentiality goals:

$$\mathrm{Kv}_{ag}(x) \equiv \exists v : \mathrm{Knows}_{ag}(x = v)$$

This particular construct expresses that agent *ag* currently knows the value of state variable x. A very first specification pattern for confidentiality goals was introduced in [vL04] based on this construct:

Goal Avoid [SensitiveInfoKnownByUnauthorizedAgent]
Formal Def $\forall ag$: *Agent* ob: $Object$:
$\neg Authorized(ag, ob.Info) \Rightarrow \neg \mathrm{Kv}_{ag}(ob.Info)$

where the *Authorized* predicate is generic and has to be instantiated through an application-specific definition.

This pattern may not be adequate in various situations. It only says that *exact* knowledge of the value of some state variable is prohibited to unauthorized agents. Let us assume that we use it for specifying our confidentiality requirement *Maintain [e-PurseBalanceConfidential]*; the knowledge by some third party that the e-purse balance is between $99 and $101 would not violate the corresponding requirement specification (as only exact knowledge of the current balance is prohibited). However, in such a situation the balance would be almost completely disclosed. Such disclosure is clearly undesirable; remember that the confidentiality goal was introduced in Section 3.3.3 in the refinement of *Avoid[Buyer Attacked for e-purse]*. In this particular context, the disclosure might be sufficient for the attacker to ensure that attacking the owner of the e-purse is sufficiently worth taking the risk. We should therefore specify stronger restrictions on knowledge of e-purse balances. We come back to this in Section 4.4.

Another disclosure can arise in case a payment aborts because the balance of the e-purse is too low. This aspect of the e-purse behavior is captured by the goal *Achieve [Payment Done If Amount Validated And Balance Sufficient]*. The knowledge capabilities of the attacker define whether he can or cannot perform the reasoning and conclude that the balance is insufficient in a situation where the payment operation aborts.

The semantics of $\mathrm{Knows}_{ag}(P)$ should make it precise how agent knowledge evolves:

- When and how do assertions get in and out of the memory of an unauthorized agent?
- What do such agents know about the target system?
- Can such agents have contradictory beliefs?
- Can such agents reason on their beliefs and make deductions? In other words, does an agent's memory merely consist of a set of assertions or of its deductive closure?

- Can such agents know that time has elapsed?

Section 4.3 provides an answer to these questions in the specific context of security requirements engineering.

4.2 Background: Logics For Reasoning About Knowledge

This section introduces some background on knowledge logics, in order to provide the necessary basis for defining the semantics of the Knows operator used throughout this chapter. Our presentation here is based on [FHMV95].

The semantics of knowledge operators found in epistemic logics can be defined either by describing the semantic properties of the logical models or by stating a set of axioms. Although we are presenting the axiomatic definition of such operators, it is worth introducing the models on which knowledge operators can be interpreted, notably because Section 4.5 presents a model-checker for out extended framework. This model-checker reasons on models and not on axioms.

The model used in epistemic logics rely on the concept of *possible worlds*. The intuitive idea behind the possible world model is that beside the true state of affairs, there are a number of other possible states of affair of "worlds". Given his current information, an agent can be able to tell which of a number of possible worlds possibly describes the actual state of affairs. Possible worlds are often called *states* in literature on epistemic logics. Temporal states and possible worlds are similar concepts, they are just assignment on variables related together with accessibility relations representing either time or kowledge. In this thesis, we use the word "state" only to refer to the temporal dimension in the sense of Section 3.2.5.1.

An agent is said to know a fact ϕ if this fact is true in all the worlds he considers possible, given his current information. His current information might include observation of the actual state of affairs, knowledge of facts that are true in all possible worlds, such as axioms, domain properties and requirements that a system enforces. Intuitively, the fewer possible worlds an agent considers possible, the more he knows because worlds represent the uncertainty she has about the current world. The semantics of epistemic operators is defined in terms of these possible worlds, which are formalized in a Kripke structure.

A *Kripke structure* for one agent is a graph where nodes represent worlds. Two worlds are adjacent if the agent cannot tell whether the current world is any of the two adjacent worlds, given that the current world is one of them. Adjacent worlds are thus indistinguishable for the agent. The adjacency

relation is often called *accessibility relation* and describes the knowledge of the agent.

A model on which an assertion including epistemic operators can be evaluated is a Kripke structure. It includes a description of the current world as well as a description of all the worlds the agent considers possible, that is; a model of the knowledge of the unauthorized agent.

It is possible to represent the knowledge of several agents in a single model. In order to do this, we need to represent several accessibility relations: each of them describes the knowledge of one agent. When reasoning about a single agent model, we do not represent the ag index of the Knows_{ag} operator.

Let M be a Kripke structure defined by (S, π, A) and a set of propositional symbols P, where S is the set of all the worlds, $\pi : S \times P \rightsquigarrow \{\text{true}, \text{false}\}$ is an interpretation for the propositional symbols for all the worlds of S and $A \subseteq S \times S$ is an accessibility relation capturing the knowledge of the unauthorized agent. $(M, s) \models \phi$ denotes that assertion ϕ is true on the model M where the real world is s. An assertion ϕ is interpreted over this structure and on the world s, that is the current world as follows:

$$
\begin{aligned}
&(M, s) \models \text{Knows}(\phi) \quad \text{iff } (M, s') \models \phi \text{ for all } s' \text{ such that } A(s, s') \\
&(M, s) \models p \text{ where } p \in P \quad \text{iff } \pi(s, p) = \text{true} \\
&(M, s) \models \phi \wedge \phi' \quad \text{iff } (M, s) \models \phi \text{ and } (M, s) \models \phi' \\
&(M, s) \models \neg\phi \quad \text{iff } (M, s) \not\models \phi
\end{aligned}
$$

Different logic systems have been proposed for defining the semantics of knowledge operators found in epistemic logics [FHMV95, JAL04]. These logic systems can be defined either axiomatically or by describing the properties of the accessibility relation.

The *S5 axiomatic system* is the strongest system for modeling knowledge in that it gives agents the most powerful reasoning capabilities [FHMV95]. The S5 system is equivalent to stating that the accessibility relation is an equivalence relation (symmetrical, transitive and reflexive). The five axioms of this system are recalled in Figure 4.1, where *Knows* denotes the knowledge operator, P and Q are assertion placeholders and ag is an agent instance. The axiom schemas hold for all possible P, Q and ag.

Other systems are generally variants of S5 where one or more axioms are removed or weakened. All these variants have different properties and therefore correspond to different knowledge-related capabilities. As another example, a formal model of agents with resource-bounded reasoning capabilities is described in [JAL04].

Knowledge logics have also been combined with temporal logics such as

- **Distribution:** Agent ag can deduce knowledge from various fragments of her knowledge

$$\text{Knows}_{ag}(P) \wedge \text{Knows}_{ag}(P \rightarrow Q) \rightarrow \text{Knows}_{ag}(Q)$$

- **Knowledge generalization:** Agent ag knows all valid formulas

$$\text{if } P \text{ is valid then } \text{Knows}_{ag}(P)$$

- **Truth axiom:** Agent ag does not have false knowledge.

$$\text{Knows}_{ag}(P) \rightarrow P$$

- **Positive introspection:** Agent ag knows that she knows.

$$\text{Knows}_{ag}(P) \rightarrow \text{Knows}_{ag}(\text{Knows}_{ag}(P))$$

- **Negative introspection:** Agent ag knows that she doesn't know.

$$\neg\text{Knows}_{ag}(P) \rightarrow \text{Knows}_{ag}(\neg\text{Knows}_{ag}(P))$$

Figure 4.1: Axioms of the S5 epistemic logic system

LTL or CTL [FHMV95, JvdMRV04]. For LTL integration, the S5 axioms are strengthened with the $\Box$-operator to hold at any time. Additional axioms capture the interaction between knowledge and time, typically, by defining memory-related capabilities of agents. Several combined systems are available dependent on whether the agents have a "synchrony" capability of measuring time or not. The two most common axioms in a synchronous system are given in Figure 4.2. They are mentioned here for exhaustiveness. It is obvious that the "no learning" axiom is not desirable for modeling the behavior of attackers, as attackers generally learn sensible information while conducting attacks. The capabilities of attackers are discussed in Section 4.3. The same axioms have been formulated without the synchrony capability, but their formal definition is very tricky and of no interest in this presentation [JvdMRV04]. These extended logic system can also be defined in term of properties binding the accessibility relation representing agents' knowledge and the "next state" relation binding two successive states in the temporal sense [JvdMRV04, FHMV95].

- **Perfect Recall:** Agent ag never forgets things she knew in the past and Agent ag can measure time (synchrony)

$$\text{Knows}_{ag}(\circ P) \Rightarrow \circ \text{Knows}_{ag} P$$

- **No Learning:** Knowledge of agent ag does not increase with time.

$$\text{Knows}_{ag} P \Rightarrow \bullet \text{Knows}_{ag} \circ P$$

Figure 4.2: Axioms for temporal knowledge

4.3 What Do Unauthorized Agents Know?

To automate reasoning about (non-) disclosure of sensitive information to unauthorized agents, we need to make our Knows_{ag} construct further precise by defining the capabilities of unauthorized agents related to their knowledge (see Section 4.1).

4.3.1 Towards a most powerful attacker model

For security-critical systems the modeled capabilities of attackers should be driven by worst-case principles. To get robust requirements about confidentiality, we need to pessimistically consider an environment where malicious agents are able to make calculations and deductions, even unanticipated ones. If such agents cannot violate the confidentiality requirements, any other will not. This worst-case principle is directly inspired from the *Most Powerful Attacker model* widely used in security engineering (see Section 2.3.2).

Reasoning about such most powerful attacker model at system run time might result in a number of false positives, that is: numerous attacks might be detected even though they are not ongoing. For instance, we might imagine an intrusion detection system that lists all the knowledge available to agents and tries to assess whether the next access to information should be authorized or not, given that some confidentiality requirements should be enforced. For run time system analyses, we would need a more accurate attacker model in order to avoid being overwhelmed by false positives.

In our context of requirements analysis, we do not care about false positives because our analysis is carried on at requirements engineering time, where all possible attacks are to be considered for system robustness. It might be the case that the resulting requirements model is very expensive to implement because there might be a huge number of countermeasures

to implement in order to achieve the desired robustness. This issue is not considered here, because we have chosen to focus on security-critical applications.

4.3.2 Shopping for axioms and security principles

We will use the most powerful attacker principle to choose axioms that further define the Knows_{ag} construct. We proceed by selecting axioms from the S5 system for defining the reasoning capabilities and well-known security principles from Section 2.3.4 for defining what an agent might directly monitor in a system. We select axioms from S5 because it is the strongest system for modelling knowledge in that it gives agents the most powerful reasoning capabilities [FHMV95]. The selected axioms are presented here.

- **Perfect System Knowledge** *Unauthorized agents know all the requirements the software implements, all the properties of the domain and all the assumptions about the environment.*

  In the e-purse system, the seller knows that payment is denied in case of insufficient balance because she knows that the system implements the functional requirement *Achieve[Payment Done Except If Insufficient Balance]*.

  This axiom implements *Kerkhoffs' principle*. This principle has been introduced in Section 2.3.4. The principle states that the security of a system should be based on the strength of its protection mechanisms, not on the obscurity of its design (see Section 2.3.4.3).

- **Maximal Input** *An agent at any time knows the value or past values of any state variable that is not explicitly specified as confidential to her at this time.*

  The agent at any time knows any past value of any state variable if that past value is not explicitly specified as confidential in the current state. In our e-purse system, the seller knows the amount that was entered, the amount that was validated, etc. This axiom ensures that the requirements model has no implicit assumptions on the confidentiality of variables.

- **Distribution** (taken from S5) *Unauthorized agents are able to make deductive inferences.*

  In our example, if a payment is denied due to insufficient balance in the e-purse, the seller is able to infer that the balance of the e-purse is lower than the amount entered in the terminal in view of her background knowledge of the requirements implemented by the system and of her observation that payment did not occur.

- **Knowledge generalization** (taken from S5) *Unauthorized agents know all tautologies.*

  In particular, we can use all these tautologies as a basis for making deductions.

  In our example, the seller knows that the assertion

  $$(e.Balance \geq \$5 \wedge \text{true}) \rightarrow \text{false}$$

  reduces to

  $$e.Balance < \$5$$

  given that he knows that

  $$(x > a) \equiv \neg(x \leq a)$$

- **Truth axiom** (taken from S5) *Unauthorized agents have no false knowledge.* This axiom supports our first definition of $\text{Knows}_{ag}(P)$ in terms of $\text{Belief}_{ag}(P)$, see Section 4.1.

- **Perfect Recall** (taken from Figure 4.2) *Unauthorized agents always remember facts and properties they used to know in the past.*

  In the e-purse example, the seller remembers, in the state where payment should take place, that the card was inserted in the terminal, that the amount was entered, and that this amount was validated.

- **Synchrony** (see Figure 4.2) *Unauthorized agents are able to measure time.*

  In our example, the seller notices that time has elapsed since the card was inserted and since the amount was validated.

- **Introspection** (taken from S5) *Unauthorized agents know that they know and know that they do not know.*

  This capability is taken to ensure that the accessibility relation used for modeling knowledge is an equivalence relation. It also makes unauthorized agents more knowledgeable. Practically, unautorized agent can reason on the fact that something is confidential to them, hence it can have some particularities.

  For instance, if one specifies that all account balance higher than \$500 must be confidential, and if the balance of an account is confidential to some agent without any other reason than this confidentiality requirement, then he can deduce that this balance is higher than \$500.

- **Instance Knowledge** *Unauthorized agents know about the existence of every instance of every object in the modeled universe.*

  Depending on the formal framework in use, this capability might or might not be a logical consequence of introspection [FHMV95]. This capability is added to the model for clarity and because it makes the attacker more knowledgeable.

- **Closure assumption** *All the knowledge of unauthorized agents directly or indirectly comes from the preceding axioms.*

  This axiom is taken to keep models realistic enough by avoiding an over-powerful attacker model. We need it when we want to show that confidentiality requirements cannot be violated within the requirements model considered.

  Intuitively, it prevents the occurrence of "deus ex machina" where the attacker gets to know sensitive information in an unexplainable way.

The closure assumption and the "Perfect System Knowledge" axiom together make the axiomatic system non-monotonic [HM85, Eng98], that is, we can have

$$P \models Q \text{ and } P \wedge R \not\models Q$$

A confidentiality requirement might be satisfied by a requirements model but violated by some extended model where, for example, some functionality has been added.

As an example, consider the signbox used in the e-signature system. User might be asked to dial their PIN code in singboxes. Suppose first that the requirements P implemented by signboxes ensures the confidentiality goal Q specifying that dialed pin code are kept confidential. If we add to signboxes the functionality R that dialed numbers are confirmed to the user through some vocal system, the new signboxes, whose acceptable behavior is modeled as $P \wedge R$ do not enforce confidentiality goal Q

Confidentiality requirement might be satisfied by the requirements model but not by the final product due to information leaks introduced at implementation time [Jac89].

4.4 Specification Patterns For Confidentiality Requirements

Section 4.1 introduced the Kv_{ag} construct that captures knowledge of the exact value of some state variable. Although appropriate in some situations, confidentiality of exact values may be too weak in others, as illustrated in

the requirement on e-purse balances. In that specific example, agents who are not the card holder should not know that there is at least $\$v$ in the e-purse (whatever the value of v is).

Recall that this goal was introduced in the refinement of *Avoid[Buyer Attacked for e-purse]*. What hurts is if potential attacker knows that the balance of some e-purse is above some "interesting" threshold. In such situation, attackers have the assurance of a reward if they take the risk to attack the e-purse holder.

Moreover, knowing that the e-purse contained at least $\$v$ six months ago is of no particular appeal to a potential card stealer; knowledge should refer to the current state in this case.

The confidentiality requirement should thus rather look like:

$$\begin{array}{l}(\forall ag\colon Agent, ep\colon ePurse) \\ \quad \neg Authorized(ag, ep.Balance) \\ \quad \Rightarrow \forall v \in ran(ep.Balance) : \neg \mathrm{Knows}_{ag}(ep.Balance > v)\end{array}$$

where $ran(ep.Balance)$ denotes the range of values of state variable $ep.Balance$. This formally states that unauthorized agents should not be able to know that $ep.Balance$ is higher than some value, whatever the value is.

As the lower bound of the range of $ep.Balance$ is 0, this confidentiality requirement can be simplified into:

$$\begin{array}{l}(\forall ag\colon Agent, ep\colon ePurse) \\ \quad \neg Authorized(ag, ep.Balance) \\ \quad \Rightarrow \neg \mathrm{Knows}_{ag}(ep.Balance > 0)\end{array}$$

In the spirit of [DAC98, CP99], we have explored specification patterns that could help engineers write recurring properties more easily as pattern instances. In our case the properties refer to confidentiality requirements. Such patterns can guide requirements elicitation; the analyst may browse the pattern catalog and ask herself whether instantiations to sensitive application objects yield relevant requirements for the system. Pattern instances may also be used as input to analysis tools.

Our pattern catalog was built systematically along two orthogonal dimensions:

- the degree of approximate knowledge to be kept confidential,
- the timing according to which that knowledge should be kept confidential.

For the temporal dimension, we explored the temporal operators related to future referencing: $\circ$ (next), $\Box_{\leq \ldots}$ (always until some delay has expired), $\mathcal{U}$ (until), $\mathcal{W}$ (unless), and $\Box$ (always). For the degree dimension, we browsed through the standard comparison over integer values and through the requirements models we elaborated or read about throughout this thesis. The catalog is of course extendable along those two dimensions.

The combination of patterns defined along the knowledge approximation (6) and the timing dimensions (5) yields a pattern space of 30 generic specifications to be considered during requirements elicitation. Retrieving the appropriate pattern from the catalog is fairly simple; the degree of approximate knowledge is determined first and then the required timing is selected.

Our pattern library exclusively focusses on specifying that a single variable should be confidential. There might be interesting extensions related to the confidentiality of facts involving several variables. This is left as an open issue for future work.

4.4.1 Degree of approximate knowledge

Patterns along this dimension include the following.

- **Confidential exact value** An agent should not know the exact value of some state variable:

$$\begin{aligned}&\text{val-Confidential}_{ag}(x) \equiv \\ &\quad \forall v \in ran(x) : \neg\text{Knows}_{ag}(x = v)\end{aligned}$$

  This is the pattern proposed in [vL04].

- **Confidential lower/upper bound** An agent should not know that the value of some state variable is below/above some threshold:

$$\begin{aligned}&\text{lb-Confidential}_{ag}(x) \equiv \\ &\quad \forall v \in ran(x) : \neg\text{Knows}_{ag}(x > v) \\ &\text{ub-Confidential}_{ag}(x) \equiv \\ &\quad \forall v \in ran(x) : \neg\text{Knows}_{ag}(x < v) \\ &\text{betw-Confidential}_{ag}(x) \equiv \\ &\quad \forall v_1, v_2 \in ran(x) : \neg\text{Knows}_{ag}(v_1 < x < v_2)\end{aligned}$$

  The confidentiality requirement *Maintain [e-Purse Balance Confidential]* is a typical example of confidentiality requirement to be made more precise by instantiating the lb-Confidential pattern. Recall

that this goal was introduced in Section 3.3.3 in the refinement of *Avoid[Buyer Attacked for e-purse]*. The idea is that potential attackers are much more likely to be tempted by attacking the e-purse owner if they are guaranteed to get a minimal return.

The amount of cash stored in a bank ATM is subject to the

$$\text{betw-Confidential}_{ag}(x)$$

confidentiality pattern in order to prevent attackers from infering a lower bound to the amount of cash retrieved by a consumer at the ATM.

The number of strategic weapons available to a nation for reprisal to an attack is subject to the ub-Confidential$_{ag}(x)$ confidentiality pattern in order to make potential reprisal look more menacing.

- **Fully confidential value** An agent should not be able to infer any property about the value of some state variable (for example, lower/upper bound, parity, order of magnitude, etc.). In other words, the agent should not be able to exclude any value within the variable's range:

$$\begin{aligned}&\text{full-Confidential}_{ag}(x) \equiv \\ &\qquad \forall v \in ran(x) : \neg\text{Knows}_{ag}(x \neq v)\end{aligned}$$

This is the strongest confidentiality requirement because it subsumes all the other degrees of confidentiality presented in this section.

Example of use: the PIN code used for user authentication at bank ATM is subject to this fully confidential requirement because there is always a probability that an attacker dials the correct pin code by chance. The full confidentiality of the PIN code contributes to keeping this probability as low as possible.

- **Confidentiality of order of magnitude** An agent should not know the order of magnitude of some state variable. In other words, the agent will always consider that the order of magnitude of that variable can be any possible value. The order of magnitude is defined as the decimal logarithm rounded down. We have arbitrarily fixed the base of the logarithm to 10. The pattern is then:

$$\begin{aligned}&\text{om-Confidential}_{ag}(x) \equiv \\ &\qquad \forall v \in 0..\lfloor log_{10}(max(ran(x))\rfloor : \\ &\qquad\qquad \neg\text{Knows}_{ag}(\lfloor log_{10}(x)\rfloor \neq v)\end{aligned}$$

4.4.2 Timing of confidentiality

To be fully precise, the engineers should also make it explicit how long should confidentiality be enforced. In this section, we specify how long a value should be kept confidential. The confidentiality pattern specifies that the present value of the variable should be kept confidential. The value of a state variable might change with time. In order to specify that all these values are confidential, the confidentiality pattern should be nested inside temporal referencing.

- **Confidential now** In the current state, an agent should not know about some state variable:

$$\text{Y-Confidential-now}_{ag}(x) \equiv \text{Y-Confidential}_{ag}(x)$$
$$\text{with } Y \in \{\text{val}, \text{lb}, \text{ub}, \text{betw}, \text{full}, \text{om}\}$$

  This is also referred to as *instantaneous confidentiality*. Example of use: In our e-purse system, we might only be concerned by unauthorized agents knowing the value of an e-purse balance in the current state where the balance has that value - without necessarily caring for such agents to know tomorrow that the balance had that value today. The confidentiality goal for the ePurse balance is therefore defined as follows:

$$(\forall ag\text{: } Agent, ep\text{: } ePurse)$$
$$\neg Authorized(ag, ep.Balance)$$
$$\Rightarrow \text{lb-Confidential-now}_{ag}(ep.Balance)$$

  Notice that there is an "always" operator hidden in the $\Rightarrow$ symbol. This operator specifies that the confidentiality pattern should be enforced on all successive values of the state variable.

- **Confidential until expiration date** An agent should not know about the value of the state variable until some delay has expired:

$$\text{Y-Confidential-upTo}_{ag,d}(x) \equiv$$
$$\forall v : (x = v) \rightarrow \Box_{\leq d}\text{Y-Confidential}_{ag}(v)$$
$$\text{with } Y \in \{\text{val}, \text{lb}, \text{ub}, \text{betw}, \text{full}, \text{om}\}$$

  Example of use: A classified NSA archive must be kept confidential for 60 years.

- **Confidential unless/until condition** An agent should not know about some state variable unless or until some condition becomes true:

$$\text{Y-Confidential-unless}_{ag,Cond}(x) \equiv$$

$$\forall v : (x = v) \rightarrow \text{Y-Confidential}_{ag}(v)\ \mathcal{W}\ Cond$$

$$\text{Y-Confidential-until}_{ag,Cond}(x) \equiv$$

$$\forall v : (x = v) \rightarrow \text{Y-Confidential}_{ag}(v)\ \mathcal{U}\ Cond$$

$$\text{with } Y \in \{\text{val}, \text{lb}, \text{ub}, \text{betw}, \text{full}, \text{om}\}$$

Example of use: In a paper publication process, submissions must be kept confidential unless they are accepted.

- **Confidential forever** An agent should never know about some state variable:

$$\text{Y-Confidential-forever}_{ag}(x) \equiv$$

$$\forall v : (x = v) \rightarrow \Box\text{Y-Confidential}_{ag}(v)$$

$$\text{with } Y \in \{\text{val}, \text{lb}, \text{ub}, \text{betw}, \text{full}, \text{om}\}$$

Example of use: In some banking applications, at all times, account balances may be required to be kept confidential forever. This is defined as:

$$(\forall ag\text{: } Agent, acc\text{: } Account)$$

$$\neg Authorized(ag, acc)$$

$$\Rightarrow \text{Y-Confidential-forever}_{ag}(acc.Balance)$$

In the informal definition, "at all times" is translated into the always operator hidden in the $\Rightarrow$ operator, and "forever" is captured in the confidentiality pattern.

4.5 Checking Confidentiality Claims on Requirements Models

We have built a proof-of-concept tool that checks requirements models against confidentiality claims that are expressed in terms of our specification patterns. The tool is named *CONCHITA* (CONfidentiality CHecker for Incremental Threat Analysis). This kind of problems solved by CONCHITA is called a *confidentiality-checking problem*.

It combines bounded temporal model checking and finite instantiation techniques to translate the confidentiality-checking problem into a constraint satisfaction problem [BCC+03, Jac00]. The latter is solved by constraint programming techniques [Sch02b].

The tool is given a requirements model including confidentiality claims. The tool generates counter-example traces leading to violation of confidentiality requirements, if any. At the end of such traces, some agent gets knowledge that violates the confidentiality claim. This knowledge is gained according to the axiomatic system describing the capabilities of the unauthorized agent regarding his knowledge. The tool also explains the reasoning performed by this unauthorized agent that delivers him the forbidden knowledge.

The difference from standard bounded model checking is that (a) the system is described at a more abstract level in terms of temporal logic assertions (instead of operational state machines), (b) the tool reasons on epistemic logic constructs and (c) it combines this technique with finite instantiation.

In this section, we present our prototype and illustrate it on the e-purse system. Section 4.5.1 presents the inputs and outputs of CONCHITA . Section 4.5.2 presents the inside of CONCHITA . Section 4.5.3 discusses the properties of CONCHITA .

4.5.1 CONCHITA as seen from outside

The following **inputs** are provided to the tool.

- A set of requirements describing the system. These are taken as system specification. They are included in the knowledge the unauthorized agent has about the behavior of the system, according to the *Perfect System Knowledge axiom*. For instance, we can submit the following requirement:

  Goal Achieve[Payment When Amount Encoded And e-Purse Inserted Except If Balance Insufficient]
  Formal Def$(\forall e$: e-purse, t : Terminal)
  $$\begin{aligned} &\text{Inserted}\langle\text{docked: } e \text{ docker: } t\rangle \\ &\wedge t.\text{amountValidated} \\ &\wedge t.\text{encodedAmount} \leq e.\text{balance} \\ &\Rightarrow \\ \circ\text{Payment}\langle&\text{from: } e,\ \text{to: } t,\ \text{amount: } t.\text{encodedAmount}^{-}\rangle \end{aligned}$$

  We can also include a set of confidentiality requirements defined through our confidentiality patterns. These are assumed to be true. A variable that is assumed to be confidential is hidden to unauthorized agents. He might still be able to deduce its value, but he is unable to directly monitor it.

- A set of confidentiality requirements, taken as claims. These requirements are defined through our confidentiality patterns, similarly to

assumed confidentiality requirements. So far, CONCHITA only supports the most typical patterns, that is, the full-Confidential-now, full-Confidential-until and full-Confidential-forever patterns. CONCHITA will try to break any of these confidentiality requirements.

For our running example, we might submit the following confidentiality claim:

Goal Maintain[e-purse balance confidential]
Formal Def($\forall$ ep: e-Purse, ag: Agent)
$\neg$Owns<owner: ag, owned: ep> $\wedge$ ag $\neq$ ep
$\Rightarrow$ full-Confidential-forever$_{\text{ag}}$(ep.balance)

This confidentiality requirement is too strong compared to the definition given in Section 4.4. This too strong definition is chosen to ensure a confidentiality violation can be found.

CONCHITA only evaluates the knowledge of agents at the end of the trace. A confidentiality claim specifying that the value of a state variable at time 1 should be instantaneously confidential cannot be evaluated on a trace of length 2. However, a confidentiality claim specifying that the value of a state variable should always be instantaneously confidential will only be evaluated and possibly violated at the end of the trace.

There can be several confidentiality claims. CONCHITA tries to break each of them in turn. When CONCHITA searches for a violation to a confidentiality claim, the other confidentiality claims are assumed to hold, that is: the variables they prescribe to be confidential are assumed to be confidential, that is: hidden to unauthorized agents.

Confidentiality claims might include outer temporal referencing. The confidentiality claim presented here above includes an "always" operator, hidden in the $\Rightarrow$ symbol. In this case, they are split into atomic sub-claims, each of which specifying that the value of the state variable at a given position in time is confidential.

- An entity-relationship model that declares the objects referred to in the requirements. For our running example, we might submit the model presented in Figure 4.3.

  CONCHITA supports boolean and integers as base types. Integers are restricted to positive values only. It is possible to declare additional base types, either as enumerations or as unbounded base type, which are similar to the ones available in the Z language [PST91].

- Directives for instantiating the object model in order to propositionalize the requirments. For example, we might bound our model to

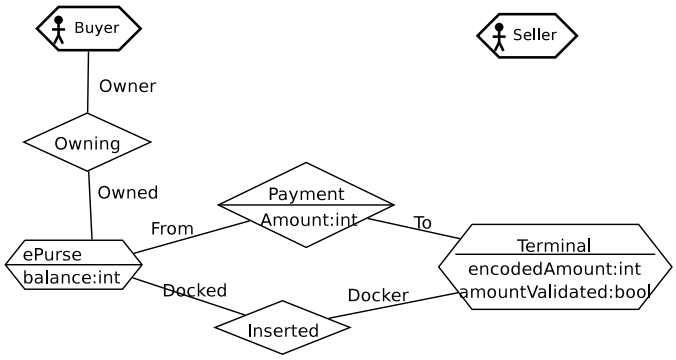

Figure 4.3: Object model fragment for the e-purse payment system

one single instance of Buyer with name "alice", one single instance of Seller with name "bob" and one single instance of Terminal:

```
Buyer:[alice]
Seller:[bob]
Terminal:1
```

- A maximal length for the output trace. CONCHITA will only search for traces with a length lower or equal to this maximal length. In the example, we take this maximal length as two.
- Optional hints for constraining the output trace to make it more "interesting". For example, the user might require that the balance of e-purses is a least 3, to avoid simple traces with zero-balance.

The BNF grammar of CONCHITA is shown in Annex A.1.

CONCHITA displays a counter-example trace, if one is found, or a message "no confidentiality violation found". The output trace leads to disclosure of one of the claimed-to-be-confidential state variables. Disclosure takes place at the last position of the trace. It is characterized by the following items:

- the state variable that was claimed to be confidential,
- the agent instance that was claimed to know nothing about this variable,
- a position in the trace and a value for the variable such that, at the end of the trace, the agent knows that the variable is different from this value at that position.

The counter-example trace delivered for our running example is shown in Figure 4.4. A graphical comic-strip-like representation of this trace is shown in Figure 4.5. The trace is displayed as a sequence of states. Each state shows the value of the attributes of entity/agent instances together with their associations. A bracketed attribute in the trace expresses that its value was assumed to be confidential and thus hidden to the agent instance (see e-Purse1.Balance in state 1).

The generated trace is actually a fragment of model, in the epistemic sense (see Section 4.2), as it only includes the actual world and not the other worlds the unauthorized agent considers possible. However, CONCHITA ensures that the other worlds are such that the knowledge of the agent violates the confidentiality claim. In the trace presented in Figure 4.4, CONCHITA ensures that there is no trace the unauthorized agent consider possible where the value of the e-purse at time 1 is 4.

```
---------------------------------------- State 1
alice (Buyer agent){}
bob (Seller agent){}
e-Purse1 (e-Purse agent){
    ** Balance: 3 ** (DISCLOSED) }
Terminal1 (Terminal agent){
    amountValidated: true
    encodedAmount: 4 }
Inserted<docked: e-Purse1
         docker: Terminal1>
Owns<owned: e-Purse1
     owner: alice>
---------------------------------------- State 2
alice (Buyer agent){}
bob (Seller agent){}
e-Purse1 (e-Purse agent){
    [ Balance: 3 ] (HIDDEN)}
Terminal1 (Terminal agent){
    amountValidated: false
    encodedAmount: 0}
Owns<owned: e-Purse1
     owner: alice>
At this state, bob knows that e-Purse1.Balance is
different from 4 at time 0
------------------------------------------------
```

Figure 4.4: Counter-example trace produced by CONCHITA

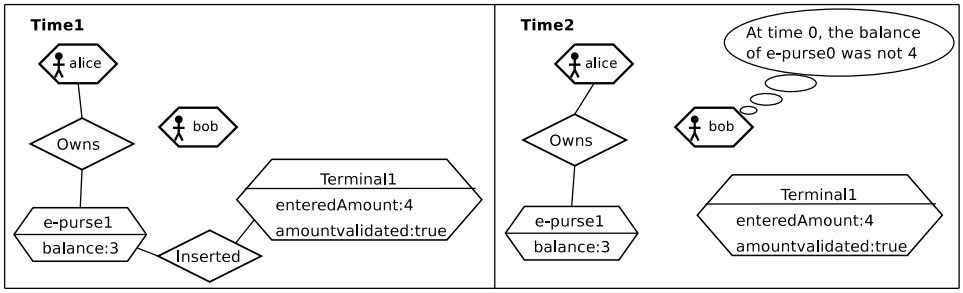

Figure 4.5: Graphical view of the output trace

As mentioned before, unviolated confidentiality claims are assumed to be enforced and confidentiality claims are split into confidentiality claims about values in the traces. The claim is split into one confidentiality claim per position in the trace because of the "always" operator implicit in the strong implication of the claim. The stars around *ePpurse1.Balance* in state 1 express that this state of the variable is disclosed at the end of the trace. The unauthorized agent is bob. At the end of the trace, he gets to know that the balance of alice's e-purse was different from 4 at time 1 (see the message generated in state 2 in Figure 4.4 or the bubble at state 2 in Figure 4.5).

Bob's reasoning goes as follows: (a) in state 1, the *e-Purse1* is inserted in *Terminal1*, the encoded amount is 4, and this amount is validated, a payment should therefore take place at the next state except if the balance of *e-Purse1* is lower than 4; (b) in state 2, no payment takes place; hence the balance of *e-Purse1* can be inferred to be strictly lower than 4, hence different from 4.

CONCHITA also provides an explanation of the reasoning performed by the unauthorized agent that enabled him to infer that the claimed-to-be-confidential variable is different from the given value. The explanation consists of a minimal set of knowledge fragments about the system that are necessary to perform the deduction. Knowledge fragments include requirements and domain properties (including association multiplicities), in which universal quantifications over objects and time have been instantiated to corresponding object instances and trace positions, respectively.

Figure 4.6 shows the explanation displayed for Bob's reasoning. In this case it includes one fragment only, taken from the requirement *Achieve[Payment When Amount Encoded And e-Purse Inserted Except If Balance Insufficient]*. In this fragment, the universally quantified variables have been instantiated to Teminal1 and e-Purse1, respectively. The "always" operator hidden in the strong implication ("⇒") has been instantiated to state 1, as mentioned on the left of the "⊨" symbol. The ˆ suffix is an ASCII notation for the $^-$ suffix used for reference to the previous value of

the state variable like in the requirements definition.

```
The unauthorized agent used the following
knowledge fragments in his inference.

Source: Achieve[Payment When Amount Encoded
                And e-Purse Inserted
                Except If Balance Insufficient]
Instantiation: [term/Terminal1, ep/e-Purse1]

1 |= Inserted<docked:e-Purse1 docker:Terminal1>
     & Terminal1.amountValidated
     & Terminal1.encodedAmount =< e-Purse1.Balance
     -> o Payment<amount: Terminal1.encodedAmount^
                  from: e-Purse1
                  to:Terminal1>
```

Figure 4.6: Explaining the reasoning for disclosure

CONCHITA only evaluates the knowledge of the unauthorized agent at the last position of the trace. Claims are evaluated and possibly detected as violated at this position only.

4.5.2 Inside CONCHITA

This section presents the internals of CONCHITA . Section 4.5.2.1 presents the formal problem solved by CONCHITA . Section 4.5.2.2 presents how this problem is reduced to a problem that can be solved by constraint programming techniques through propositionalization and bounded model checking techniques. It also presents the overall architecture of CONCHITA . Section 4.5.2.3 to Section 4.5.2.7 present the different parts of this architecture.

4.5.2.1 Basic Principle

The basic idea behind CONCHITA is that a confidentiality-checking problem can be translated into an equivalent constraint satisfaction problem. This requires our confidentiality patterns to be encoded as quantifications over traces envisioned by the unauthorized agent, given (i) his knowledge of the system and (ii) his knowledge of the variables that are not claimed nor assumed to be confidential.

Suppose there is only one confidentiality claim on one value of one state variable x. Let H be the system behavior as described by the conjunction

of the functional requirements and domain properties submitted to CONCHITA . Let F be the conjunction of the additional hints submitted to CONCHITA . Let T be a trace searched by CONCHITA . The confidentiality problem solved by CONCHITA can be paraphrased as follows:

find a value c in the range of the claimed confidential state variable x
and a trace T **such that**:
(a)T $\models H \wedge F$
(b)there is no trace T' **such that**:

$$\left(\begin{array}{l} T' \models H \\ \wedge\text{non-confidential variables} \\ \qquad\text{have same values in } T \text{ and } T' \\ \wedge\text{claimed-confidential variable } x \text{ in } T' \\ \qquad\text{is equal to } c \end{array}\right)$$

Constraint (a) expresses that the trace T found must be admissible with respect to the system behavior as described by $H \wedge F$.

Constraint (b) expresses that the agent knows that the confidential variable is different from c, given his knowledge (i.e., the system specification H, as prescribed by Kerkhoff's assumption in Section 4.3 and the value of each non-confidential variable, as prescribed by the maximal input axiom in Section 4.3). The hints represented by F are not part of the knowledge of the unauthorized agent.

Knowing that a the value of a claimed confidential variable is not c is very weak, there is not much knowledge about the variable. This knowledge is sufficient to break confidentiality claims defined with the full-Confidential$_{ag}(x)$ patterns. As discussed in Section 4.5.1, these patterns are the only ones supported by CONCHITA .

The constraint problem presented above requires model-checking techniques on first order assertions and cannot be solved efficiently. CONCHITA is made efficient by instantiation of our models to small worlds, and by looking for small traces only, as done by first order counterexample finders and by bounded temporal model checkers, respectively [BCC$^+$03, Jac00]. Thanks to these boundings, the initial confidentiality-checking problem is translated into a confidentiality-checking problem involving a finite number of variables with finite ranges. The latter can also be reduced to a constraint satisfaction problem and efficiently solved via constraint programming techniques [Sch02b].

4.5.2.2 Structuring a Technique for Solving Confidentiality-Checking Problems

CONCHITA is structured into four layers. Layers are organized into a "use" hierarchy. The top three layers translate their input problem into a simpler problem that is submitted to the layer underneath. The bottom layer performs the computation using constraint-programming techniques [Sch02b, RH04]. It also contains a constraint satisfaction diagnosis algorithm used for generating the unauthorized agent's reasoning. The solution found by the bottom layer percolates up through the upper layer; it is gradually translated into a solution to the top input problem.

The overall architecture of the tool is shown in Figure 4.7. The system specification there is made of the requirements, the domain properties and the hints. The unauthorized agent's knowledge is made of the requirements and the domain properties.

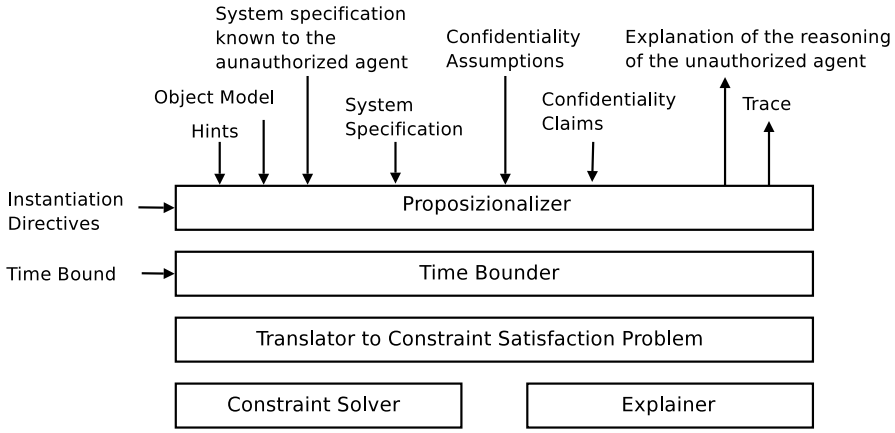

Figure 4.7: CONCHITA's Overall Architecture

Propositionalizer The first order input model is propositionalized by instantiating the object model according to the directives of the input file. We also switch to a mono-agent model where we consider that there is only one unauthorized agent.

Time Bounder The length of the trace is determined by user input and the temporal confidentiality-checking problem is translated into a non temporal confidentiality-checking problem. CONCHITA will only look for trace of size lower or equal to the number of time units prescribed. It generates an equivalent problem for each trace length and submit it to the lower layer. The size of the trace is gradually extended until the maximal length is reached or a trace is found.

Translator to a Constraint Satisfaction Problem The non-temporal confidentiality-checking problem is translated into an equivalent constraint satisfaction problem. The problem produced by this layer is expressed without any reference to epistemic logics. The delivered constraint satisfaction problem includes a universal quantification on variables that are hidden to unauthorized agents.

Constraint Solver The constraint satisfaction problem is solved by means of constraint programming techniques. The universal quantification introduced in the constraint satisfaction problem asks for dedicated algorithms.

Explainer The explainer algorithm computes the minimal set of knowledge fragments that are necessary for the unauthorized agent to make the deduction violating the confidentiality claim.

The solution found at the lowest layer is an interpretation for the non-temporal variables. This solution is then translated into a solution to the initial confidentiality-checking problem by upwards propagating the solution along the layers. We do not detail this upward process here as it can trivially be deduced from the downwards process.

The purpose of the propositionalization and time bounding layers is to translate the input confidentiality-checking problem into a confidentiality-checking problem with a finite number of variables, the latter ranging over finite domains. This condition must be fulfilled in order to solve the resulting confidentiality-checking problem by efficient constraint solving techniques.

The propositionalization and time bounding layers do not preserve the semantics as they restrict the expressiveness of the model. To ensure tool soundness, we therefore *strengthen* the translation of the system specification and confidentiality assumptions while *weakening* the translation of the confidentiality claims and of the knowledge of the unauthorized agent. Such dual translations are known as *expanding* and *contracting* abstractions [Pnu98]. Intuitively, it means that we will be more demanding on generated traces while unauthorized agents will know a little bit less about the system. For example, when there is a requirement of the form "$\Diamond B$" it is going to be translated into "$\Diamond_{\leq \text{`end of trace'}} B$" for the system description and "true" for the knowledge the unauthorized agent has about the behavior of the system, respectively. Knowing that "true" holds does not any knowledge to the unauthorized agent because he knows all the tautologies. With such translation, we ensure that the delivered trace enforces the initial requirement "$\Diamond B$" and that the attacker does not have more knowledge than the one he is supposed to have within an "unbounded"-time world.

The constraint satisfaction problem solved by the bottom layer of CON-

CHITA is the same as the one presented at the beginning of the section except that

- The trace is made of a finite and pre-determined set of variables (thanks to propositionalization and time-bounding),
- The constraint (a) is replaced with the following constraint: $\mathrm{T} \models H_s \wedge F_s$. The indices s denotes that these are obtained by propositionalization and time bounding with strengthening. Under some conditions discussed in Section 4.5.2.3 and Section 4.5.2.4 these transforms ensure that
$$\mathrm{T} \models H_s \wedge F_s \text{ implies } \mathrm{T} \models H \wedge F$$
- The first fragment of constraint (b) is replaced with $T' \models H_w$. The indices w denotes that these are obtained by propositionalization and time bounding with weakening. Under some conditions discussed in Section 4.5.2.3 and Section 4.5.2.4 we have that $T' \models H$ implies $T' \models H_w$.

The two last bullets ensure that the solution to the constraint satisfaction problem found by CONCHITA at the lower layer is a solution to the confidentiality-checking problem presented at the beginning of this section.

4.5.2.3 Propositionalizing the Input Model

The goal of this layer is to translate the first order temporal confidentiality-checking problem into an "equivalent" form with a finite number of temporal variable ranging on a finite domain. These domains are the domains of the attributes of entities, agents and associations. These attributes can be declared as booleans, finite enumerated types or integers.

The propozitionalization layer translates a first order temporal confidentiality problem into a temporal confidentiality problem. This is achieved by propositionalizing the object model by declaring a finite set of instances for each of the declared objects class. This technique is the one used by the Alloy analyzer to check first order theorems written in the Alloy language [Jac00, Jac06].

Integers must be translated to range over a finite domain. However, they cannot be bounded to a finite range such as $[0, \ldots, 5]$ (recall that integers are positive in the input language of CONCHITA , hence the 0 is not an additional bound). This trivial bounding technique inserts an additional constraint that "every integer is lower than 5". If we were to bound integers in this way, the unauthorized agent might be able to deduce false knowledge.

For instance, he would know that the balance of every e-purse is always lower or equal to 5 and perform arithmetics based on such erroneous knowledge.

In order to prevent this, integers are bounded to a finite number of values that contains an abstraction for all the other values, that is, they are bounded into the finite range: $\{0, 1, ..., n, \text{more}\}$ where "more" represents all the other values for integers. This abstraction imposes that all operators on integers ($<, \leq, =, +, *$) must be translated to handle this abstract value. These operators must be translated with strengthening or weakening, depending on the context (description of the behavior of the system or of the knowledge the unauthorized agent has about this behavior) and on whether they appear positively or negatively (within an odd or even number of negations). This technique for bounding integers is fully documented in Annex A.3.

Due to the *instance knowledge* axiom defining our attacker model in Section 4.3, the attacker knows about the existence of every instance of every object class. We can therefore provide the same object model bounding when reasoning about the knowledge of the unauthorized agent as when reasoning about the behavior of the system. By doing this, our model of attacker is given the knowledge that he is reasoning about a bounded universe and he knows the bounds, that is: he knows every instance of every object in his universe. The finite instantiation process fixes the number of instances within the world of the unauthorized agent, hence, this information is also added to the unauthorized knowledge the unauthorized agent has about the current world.

Once the object model is finitely instantiated, it is possible to produce a finite set of variables that represent all the possile models with this number of instances [Jac00]. Given a finite instantiation for an object model, CONCHITA creates a finite set of state variables representing the attributes and features of each of the instances.

For instance, the the e-purse system, the *e-Purse* agent class has one attribute: *balance* with type *int*. Therefore, CONCHITA creates one state variable in order to represent the balance of the single instance of e-Purses. This state variables is denoted *e-Purse1.balance*.

Associations are propositionalized similarly to agents and entities except that each instance has an additional boolean attribute *Alive* that expresses whether or not the association instance holds with the corresponding field values. This allows the number of holding associations to be modulated in a propositionalized history [vL03a].

The assertions are translated accordingly to these finite instantiations:

- Existential and universal quantifiers are translated as conjunctions or

disjunctions where the quantified variable is instantiated to the finite number of instance declared for the range of the quantification;

- Attribute selection for entities and agents are replaced with reference to the corresponding state variables;
- References to associations are replaced with a disjunction over the finite members of the association type. For example, on the basis of such instantiation, the assertion:

$$\text{Inserted<docker : Terminal1 docked : e-purse1>}$$

is translated into the following:

$$\left(\begin{array}{l}\text{Inserted1.alive}\\ \wedge\text{Inserted1.docker} = \text{Terminal1}\\ \wedge\text{Inserted1.docked} = \text{e-purse1}\end{array}\right) \vee \left(\begin{array}{l}\text{Inserted2.alive}\\ \wedge\text{Inserted2.docker} = \text{Terminal1}\\ \wedge\text{Inserted2.docked} = \text{e-purse1}\end{array}\right)$$

The confidentiality patterns also ask some translation at this layer because they all refer to an agent. For instance, the confidentiality goal of the running example contains the following assertion:

$$\text{full-Confidential-forever}_a(p.\text{balance})$$

where a is a universally quantified variable on agent instances and p is a quantified variable on ePurse instances. If we remove the object structuring from the model, we need to somehow encode the reference to agents into the produced model.

The idea is to switch from a multi-agent model into a mono-agent model, that is, remove the agent indices from the confidentiality patterns. It is also possible to keep a multi-agent model for knowledge modeling but, as we are removing the object structuring, we prefer to remove all references to the object model at once.

It is possible to switch to a mono-agent model because, though the input model contains several agents, we only reason about the knowledge of one unauthorized agent. This unauthorized agent might be any of the modeled agent instances. We capture his identity with an extra variable. This variable is denoted *UnauthorizedAgent* and ranges over the names of the declared agents. As the unauthorized agent does not change during the trace, the *UnauthorizedAgent* variable is rigid, i.e.: its value does not change along the trace. Using this variable, the assertion:

$$\text{full-Confidential-forever}_{\text{Bob}}(\text{ePurse1.balance})$$

is translated as follows:

$$\begin{aligned} &\text{UnauthorizedAgent} = \text{'Bob'} \\ &\quad \rightarrow \text{full-Confidential-forever}(\text{ePurse1.balance}) \end{aligned}$$

Every confidentiality pattern in the problem declaration refers to the same agent. The reference to this agent is therefore dropped from the confidentiality patterns. In the lower layers, we will only reason on this single agent.

CONCHITA also supports an extra option to insert an additional agent representing all the external agents, thus the ones that do not appear in the object model and that will 'see' the system from an external point of view.

4.5.2.4 Bounding Time

The goal of this layer is to translate the propositional temporal confidentiality-checking problem into an "equivalent" form with a finite number of non-temporal variables. This layer ensures that a solution to the latter can be translated into a solution to the former one.

The translation is performed by bounding time to a fixed value. We therefore only look for traces of a fixed length. For instance, example, we set the length of searched traces to 2. CONCHITA supports an option where it looks for traces up to the maximal length. In that case, it searches for a traces with of length 1, then, in case it did not find one, for a trace of length 2, and so on until the maximal length is reached. The first discovered trace is returned and ends the search for trace.

This bounding technique is widely used in the context of state-machine model-checking. It is known as bounded model checking [BCC+03, BCC+99, HR00].

The input problem is made of five parts. They all include references to time. They are listed below.

- The model: the model is a set of temporal variables.
- The definition of the behavior of the system. This behavior is specified by the requirements and hints. They include temporal referencing.
- The definition of the knowledge the unauthorized agent has about the behavior of the system.
- The confidentiality claims defined by confidentiality patterns. These patterns can be nested in temporal referencing and include temporal referencing.

- The assumed confidentiality requirements, which are defined in a similar way than confidentiality claims.

In order to create a model without time referencing, we need to translate time referencing into something else for these five parts.

In order to remove time referencing from the model, we create a set of non-temporal variables representing the value of temporal variables at every position in the trace. A non-temporal variable represents the value of a temporal variable at a certain position in the trace. As the length of the trace is finite and the number of temporal variables is finite, the number of non-temporal variables is also finite.

In order to remove time referencing from the description of the behavior of the system and from the knowledge the unauthorized agent has of this behavior, we unfold the temporal operators on the finite trace and replace references to temporal variables at the position in time given by the unfolding with a reference to the corresponding non-temporal variable.

In order to remove time referencing from the confidentiality claims, and from the assumed confidentiality requirements, we evaluate the knowledge of the unauthorized agent at the end of the trace. The "Perfect Recall" and "maximal input" axioms from our most powerful attacker model are used here, together with the assumed confidentiality goals to define which values of the trace the attacker knows at the end of the trace. Some confidentiality claims must be dropped here, as explained in Section 4.5.1, because we can only evaluate confidentiality claims claiming that a given variable is confidential at the end of the trace.

The remaining of this section presents how time referencing is removed from these five parts of the model, according to the principles presented here above.

Capturing the values of temporal variables in finite traces into non-temporal variables

A set of non-temporal variables is created in order to represent the value of each state variable at each position in the trace. For instance, the state variable $EPurse0.balance$ is represented by two non-temporal variables, namely $EPurse0.balance_0$ and $EPurse0.balance_1$. The subscripted numbers denote the position in the trace. These two non-temporal state variables represent the value of their corresponding temporal state variable at time 0 and 1, respectively.

All temporal formulas must be translated according to this bounding. This translation removes all temporal referencing from the input model. This is achieved by replacing references to state variables with references to non-temporal variables. For instance, consider the following assertion,

which is a hint describing the behavior of the system:

$$\Box(EPurse0.balance \geq 0)$$

It is translated in a time-bounded model of length two as follows:

$$EPurse0.balance_0 \geq 0 \wedge EPurse0.balance_1 \geq 0$$

The translation is performed by applying a set of translation rules that are to be presented here. In the preceding example, we translated the formula $\Box P$ into a conjunction of P's evaluated at each position in the trace.

Removing reference to time from the description of the behavior of the system

The translation of the temporal assertions describing the behavior of the system into a non-temporal assertion should enforce the soundness property that any finite trace enforcing the translated assertion should be a prefix of an infinite history enforcing the temporal assertion. This translation rule is referenced to as the "translation with strengthening", to distinguish from the "translation with weakening" we will use for translating the knowledge that agent has about the behavior of the system. The desired property of the bounding with strengthening is defined as follows:

$$\sigma \models_d \text{Bound}(\Phi, d, strong) \quad \text{implies} \quad (\exists \pi)\sigma|\pi \models \Phi \tag{4.1}$$

where σ is a finite trace of length d (that is equivalent to an assignment to the non-temporal variables), Φ is the behavior of the system, this behavior is described by the requirements and the hints, $\text{Bound}(\Phi, d, strong)$ denotes the non-temporal assertion obtained by bounding assertion Φ to a finite trace of length d with strengthening (hence the argument "*strong*", we will see below that there is a dual bounding procedure with weakening for the knowledge the unauthorized agent has about the behavior of the system) $\models_d$ denotes that the finite trace on the left is a model of the non-temporal assertion on the right, π is an unbounded history. $\sigma|\pi$ is a history resulting from the concatenation of the trace σ and the history π, $\sigma|\pi \models \Phi$ expresses that the history $\sigma|\pi$ is a model of the temporal assertion Φ.

A typical example of translation rule is the one of the $\circ$ operator. It is as follows:

$$\begin{aligned} bound(\text{"}\circ P\text{"}, time, strong) \quad \rightsquigarrow \quad & \textbf{if } time \neq end \textbf{ then} \\ & \quad bound(\text{"}P\text{"}, time + 1, strong) \\ & \textbf{else} \\ & \quad \text{false} \\ & \textbf{fi} \end{aligned}$$

where *time* denotes an integer that is the position in the trace, P is an assertion that might include temporal referencing, *end* denotes the indice of the last position in the trace, and $\rightsquigarrow$ denotes that the formula on the left reduces to the formula on the right.

According to this rule, $\circ P$ is translated into false at the last position of the trace. Intuitively, as we do not have any mean of evaluating P after the end of the trace, we fix the value of $\circ P$ to an arbitrary value that matches the principle of the translation rule, that is; strengthening.

The complete translation rule with strengthening is very common in bounded model checking [BCC+03, BCC+99, HR00]. It is presented in detail in Annex A.

As an example, we consider here the goal *Achieve[Payment Done When Amount Encoded And Card Inserted Except If Balance Insufficient]*. The temporal propositional model of this assertion was the same for the propositionalization with weakening and strengthening (except for the integer operators, which we neglect here). It is as follows:

$$
\begin{aligned}
&(\text{Inserted0.alive}\\
&\wedge \text{Terminal0.amountValidated}\\
&\wedge \text{Terminal0.encodedAmount} \leq \text{ePurse0.balance}\\
&\Rightarrow \circ\text{Payment0.alive}\\
&\quad \wedge \text{Payment0.from} = \text{ePurse0}\\
&\quad \wedge \text{Payment0.to} = \text{Terminal0}\\
&\quad \wedge \text{Payment0.amount} = \text{Terminal0.encodedAmount}^{-})
\end{aligned}
$$

The translation of this goal into non-temporal logics with strengthening is as follows:

$$
\begin{aligned}
&(\text{Inserted0.alive}_0\\
&\wedge \text{Terminal0.amountValidated}_0\\
&\wedge \text{Terminal0.encodedAmount}_0 \leq \text{ePurse0.balance}_0\\
&\rightarrow \text{Payment0.alive}_1\\
&\quad \wedge \text{Payment0.from}_1 = \text{ePurse0}\\
&\quad \wedge \text{Payment0.to}_1 = \text{Terminal0}\\
&\quad \wedge \text{Payment0.amount}_1 = \text{Terminal0.encodedAmount}_0)\\
&\wedge\\
&(\text{Inserted0.alive}_1\\
&\wedge \text{Terminal0.amountValidated}_1\\
&\wedge \text{Terminal0.encodedAmount}_1 \leq \text{ePurse0.balance}_1
\end{aligned}
$$

$\rightarrow$ false)

Representing the knowledge that unauthorized agent has about the system in a finite trace

The translation of the knowledge the agent has about the behavior of the system into a non-temporal asseryion should not add more knowledge to the attacker's model. The translation of $\circ P$ presented above clearly violates this principle because it adds the information that P is false just after the end of the trace. The attacker model in the "unbounded" confidentiality-checking problem did not know this information.

The knowledge the agent has about the behavior of the system into a non-temporal problem should enforce the property that if a finite trace is a prefix of an acceptable history, then it should be acceptable by the bounded model. We therefore provide a translation rule that translates temporal assertions into non-temporal ones with weakening. This property of the translation rule is formally defined as follows:

$$(\exists \pi)\sigma|\pi \models \Phi \quad \text{implies} \quad \sigma \models_d \text{Bound}(\Phi, d, weak) \tag{4.2}$$

where σ is a finite trace of length d (that is equivalent to an assignment to the non-temporal variables), Φ is the knowledge the unauthorized agent has about the behavior of the system $\text{Bound}(\Phi, d, weak)$ denotes the non-temporal assertion obtained by bounding assertion Φ to a finite trace of length d with weakening (hence the argument "*weak*", to differentiate from the translation rule with strengthening presented here above for translating the behavior of the system), $\models_d$ denotes that the finite trace on the left is a model of the non-temporal assertion on the right, π is an unbounded history. $\sigma|\pi$ is a history resulting from the concatenation of the trace σ and the history π, $\sigma|\pi \models \Phi$ expresses that the history $\sigma|\pi$ is a model of the temporal assertion Φ.

A typical example of translation rule is the one of the $\circ$ operator. It is as follows:

$$\begin{aligned} bound(\text{"}\circ P\text{"}, time, weak) \quad \rightsquigarrow \quad & \textbf{if } time \neq end \textbf{ then} \\ & \quad bound(\text{"}P\text{"}, time + 1, weak) \\ & \textbf{else} \\ & \quad \text{true} \\ & \textbf{fi} \end{aligned}$$

According to this rule, $\circ P$ is translated into true at the last position of the trace. Intuitively, as we do not have any mean of evaluating P after the end of the trace, we fix the value of $\circ P$ to an arbitrary value that matches the principle of the translation rule, that is: weakening.

The translation rule for translating the knowledge the unauthorized agent has about the behavior of the system into a non-temporal assertion is fully presented in Annex A.

The translation of the goal *Achieve[Payment Done When Amount Encoded And Card Inserted Except If Balance Insufficient]* appearing in the knowledge the agent has about the behavior of the system is as follows:

$$
\begin{aligned}
&(\text{Inserted0.alive}_0 \\
&\wedge \text{Terminal0.amountValidated}_0 \\
&\wedge \text{Terminal0.encodedAmount}_0 \leq \text{ePurse0.balance}_0 \\
&\rightarrow \text{Payment0.alive}_1 \\
&\quad \wedge \text{Payment0.from}_1 = \text{ePurse0} \\
&\quad \wedge \text{Payment0.to}_1 = \text{Terminal0} \\
&\quad \wedge \text{Payment0.amount}_1 = \text{Terminal0.encodedAmount}_0) \\
&\wedge \\
&(\text{Inserted0.alive}_1 \\
&\wedge \text{Terminal0.amountValidated}_1 \\
&\wedge \text{Terminal0.encodedAmount}_1 \leq \text{ePurse0.balance}_1 \\
&\rightarrow \text{true})
\end{aligned}
$$

Translating assumed confidentiality requirements

We assume here that assumed confidentiality requirements match the following pattern:

$$P \Rightarrow \text{Confidentiality} \tag{4.3}$$

where *Confidentiality* is one of the full-confidentiality patterns, and P is a propositional assertion that does not include any temporal operator. Other patterns than the simple implication are also acceptable, of course, as far as they can be translated into (4.3). CONCHITA includes an auxiliary procedure to handle this. At one whole, confidentiality patterns cannot appear negatively in the input model and cannot appear within a disjunction, including disjunction over time, such as $\Diamond$.

As discussed in Section 4.5.1 and at the beginning of this section, CONCHITA only evaluates the knowledge of the unauthorized agent at the end of the trace. Translating the assumed confidentiality requirements amounts to listing the values of the trace that the unauthorized agent knows. Such values are the values of temporal variable at each position in the trace. At the end of the trace, the unauthorized agent knows a value if there is no assumed confidentiality requirement prescribing that it is not known to him.

While translating claims and assumptions into non-temporal assertions, we also regroup the assumed confidentiality requirements and the confidentiality claims according to the variable they concern, that is: the variable assumed or claimed to be confidential. A standard way of resolving such structure clashes is to introduce an intermediate data structure [Jac75]. Two tables are associated with each variable appearing in a claim or assumption to collect the various claims and assumptions for this variable. The first table is called the *assumtion table* and the second one is called the *claim table*. They collect confidentiality claims and assumed confidentiality, respectively. Confidentiality claims in the non-temporal model are extracted from these tables.

Figure 4.8 presents an assumption table on some state variable x on a trace of length 2. Each row i collects the assumed confidentiality requirements on the value of the state variable x at a certain position i in the trace. The first column contains the position i of the value of the state variable in the trace. The second column recalls the non-temporal variable x_i that captures this value in the non-temporal model. The other columns contain the assumed confidentiality requirement $A_{x_i,j}$ on this value that hold at each position j in the trace. For instance, the assertion $A_{x_0,1}$ in the table expresses that *"the value of* x *at time* 0 *is confidential at time* 1 *of the trace"*. If there is no such assumed confidentiality requirement, the corresponding A is the constant "false". We do not consider here confidentiality on future values of state variables. As a consequence, there is no $A_{x_1,0}$ because it would express that *"The value of* x *at time* 1 *is confidential at time* 0*"*.

Time of value	**Non-temporal variable**	**Time of Confidentiality**	
		0	**1**
0	x_0	$A_{x_0,0}$	$A_{x_0,1}$
1	x_1	—	$A_{x_1,1}$

Figure 4.8: Assumption Table of State Variable x

Starting from the tables collecting the assumed confidentiality requirement, we generate the non-temporal assumed confidentiality requirement according to the following rules:

- Due to the **maximal input** axiom, a value x_i of some state variable x at position i in a trace is known by an agent at state s if it is not confidential at this state, ie, if $A_{x_i,s}$ is false.
- Due to the **perfect recall** axiom, a value x_i that is disclosed at a state s because no assumed confidentiality requirement prescribes that it is confidential at this state (ie: $A_{x_i,s}$ is false) is stored in the agent's

memory forever. It is therefore disclosed at each state t coming after s (ie: $t \geq s$).

Therefore, in a trace of length n, a value x_i of some state variable x at position i in the trace with $i \leq (n-1)$ is assumed to be confidential at the end of the trace if it is assumed to be confidential at each state from i to $n-1$. This is formally expressed in Figure 4.9. This table contains an extra column that contains the assumed non-temporal confidentiality requirement that is extracted from the line of the assumption table.

Time of value	**Time of confidentiality**					**Assumed confidentiality requirement at $t = end$ with perfect recall and maximal input**
	0	...	**i**	...	**end**	
i	—	—	$A_{x_i,i}$	...	$A_{x_i,end}$	$(\bigwedge_{j \in [i..end]} A_{x_i,j})$ $\rightarrow$ full-Confidential(x_i)

Figure 4.9: Translating assumption tables to non-temporal assumptions

In order to generate the table, we start with empty table where each cell includes the value false. Cells are gradually or-ed with additional conditions coming from assumed confidentiality requirements. We now detail the content of the cells generated by each of the three confidentiality patterns full-Confidential, full-Confidential-forever and full-Confidential-until confidentiality patterns. The translations are given below for a trace of length 2. They are easily extended to the general case.

- $\Box(P \rightarrow$ full-Confidential-now$(x))$

Time of value	**Non-temporal variable**	**Time of Confidentiality**	
		0	**1**
0	x_0	P_0	false
1	x_1	—	P_1

Where $P_i = bound(\text{"}P\text{"}, i, weak)$

- $\Box(P \rightarrow$ full-Confidential-forever$(x))$

Time of value	**Non-temporal variable**	**Time of Confidentiality**	
		0	**1**
0	x_0	P_0	P_0
1	x_1	—	P_1

Where $P_i = bound("P", i, weak)$

- $\Box(R \rightarrow \text{full-Confidential-until}_Q(x))$

Time of value	Non-temporal variable	Time of Confidentiality	
		0	**1**
0	x_0	$R_0 \wedge \neg Q_0$	$R_0 \wedge \neg Q_1$
1	x_1	—	$R_1 \wedge \neg Q_1$

Where $Q_i = bound("Q", i, strong)$ and $R_i = bound("R", i, weak)$

The translation of P_i, Q_i, and R_i ensures that the assumed confidentiality requirement is translated into non-temporal logic with a strengthening, that is: the unauthorized agent in the bounded model does not know more than the unauthorized agent in the unbounded model. This contributes to making the temporal layer sound.

When there are several assumed confidentiality requirements on the same state variable, the result of their respective translation is unified into one single table by or-ing the content of the corresponding cells in their respective tables. For instance, unifying the two last tables in the bullets above leads to the following assumption table:

Time of value	Non-temporal variable	Time of Confidentiality	
		0	**1**
0	x_0	$P_0 \vee (R_0 \wedge \neg Q_0)$	$P_0 \vee (R_0 \wedge \neg Q_1)$
1	x_1	—	$P_1 \vee (R_1 \wedge \neg Q_1)$

By applying the rule defined in Figure 4.9 on the assumed confidentiality requirements of the running example, we obtain the following assumed non-temporal confidentiality requirements:

$$\text{UnauthorizedAgent} = \text{'Bob'} \rightarrow \text{full-Confidential}(\text{EPurse0.balance}_0) \quad (4.4)$$

$$\text{UnauthorizedAgent} = \text{'Bob'} \rightarrow \text{full-Confidential}(\text{EPurse0.balance}_1) \quad (4.5)$$

Translating the Confidentiality Claims

We assume here that confidentiality claims match the pattern (4.3), like assumed confidentiality requirements. According to this pattern, confidentiality claims should be evaluated at each position in the trace, because of the outer "always" operator. However, CONCHITA only evaluates the knowledge of the unauthorized agent at the last position of the trace, as explained in Section 4.5.1.

Confidentiality claims are translated into the non-temporal model similarly except the two following differences:

- The translation of temporal claims into the tabular format is performed with *weakening*, that is: the claims in the non-temporal model are weaker than the ones in the temporal model. This contributes to making CONCHITA sound.
- The non-temporal claim extracted from the table are built only on the last column of the claim tables. This is because CONCHITA only evaluates the knowledge of the unauthorized agent at the end of the trace, as discussed at the beginning of this section. This is illustrated in Figure 4.10.

Time of value	**Time of Confidentiality**					**Confidentiality claim at $t = end$ with PR and Maximal input**
	0	...	**i**	...	**end**	
i	—	—	$C_{x_i,i}$	...	$C_{x_i,end}$	$C_{x_i,end} \rightarrow$ full-Confidential(x_i)

Figure 4.10: Translating claim tables to non-temporal claims

By applying the rule defined in Figure 4.10 on the temporal propositional assumption of the running example, we obtain the following non-temporal assumptions:

$$\text{UnauthorizedAgent} = \text{'Bob'} \rightarrow \text{full-Confidential}(\text{EPurse0.balance}_0) \quad (4.6)$$

$$\text{UnauthorizedAgent} = \text{'Bob'} \rightarrow \text{full-Confidential}(\text{EPurse0.balance}_1) \quad (4.7)$$

4.5.2.5 Expressing a Confidentiality Problem as a Constraint Satisfaction Problem

This layer translates a propositional non-temporal confidentiality problem into an equivalent constraint satisfaction problem. The solution to these two problems is an assignment to a set of non-temporal propositional variables.

Confidentiality problems fed to this layer are made of

- a set of variables over finite domains;
- a set of confidentiality claims with the pattern

$$P \rightarrow \text{full-Confidential}(\ldots)$$

where P is an assertion;

- a set of assumed confidentiality requirements with the same pattern;
- an assertion describing the behavior of the system, noted H_s;
- an assertion describing the knowledge of the attacker about the behavior of the system, noted H_w;

The output of this layer is an interpretation for the variables that is a model of the behavior of the system (H_s) and that lead to a violation of a claim, given the fact that the unauthorized agent knows every variable except the assumed to be confidential ones (due to the **maximal input** axiom) and that he knows the behavior of the system described by H_w and that he has some reasoning capabilities.

We assume that there is at most one confidentiality claim per variable and at most one assumed confidentiality requirement per variable. The algorithm used in the temporal layer enforces this assumption.

For instance, the running example matches the pattern:

$$\begin{aligned}
&\text{claim: } P(c_1, c_2, \overline{p}) \rightarrow \text{full-Confidential}(c_1)\\
&\text{claim: } Q(c_1, c_2, \overline{p}) \rightarrow \text{full-Confidential}(c_2)\\
&\text{assume: } R(c_1, c_2, \overline{p}) \rightarrow \text{full-Confidential}(c_1)\\
&\text{assume: } S(c_1, c_2, \overline{p}) \rightarrow \text{full-Confidential}(c_2)\\
&\text{assertion: } H_s(c_1, c_2, \overline{p})\\
&\text{knowledge the agent has about the behavior}\\
&\qquad\qquad \text{of the system: } H_w(c_1, c_2, \overline{p})
\end{aligned}$$

where c_1 and c_2 are shortened names for the variables ePurse0.balance$_0$ and ePurse1.balance$_1$, respectively. In our running example, P, Q, R and S are defined in (4.4, 4.5, 4.6, and 4.7). In this example, they are the same, yet, we distinguish them here because, in the general case, they might not be identical. H_s and H_w are obtained by bounding time and propositionalizing the requirements describing the system with strengthening and weakening, respectively. $\overline{p}$ is a vector representing all the other variables of the case study. These other variables are public because there is no assumed confidentiality requirement or claim on them.

This section first presents the translation rule to translate restricted form of confidentiality problems where there is only one claim and one assumed confidentiality requirement with tautological premises (the P, and R here above). This translation rule is then gradually extended by introducing a set of techniques to handle multiple claim/assumed requirements, non-tautological premises, and claim/assumed requirements on the same variables. The correctness of each translation rule is shown on the basis of

the preceding one, except the first one, which is trivial. The roadmap of this section is as follows:

1. Translating confidentiality problems with one claim and one assumed confidentiality requirement both with tautological premises. The claim and the assumed confidentiality requirement concern different variables.

2. Extending the simple translation rule to one claim and one assumed confidentiality requirement **with non-tautological premises**. The claim and the assumed confidentiality requirement concern different variables.

3. Extending the second translation rule to one claim and **several** assumed confidentiality requirements with non-tautological premises. The claim and the assumed confidentiality requirements concern different variables.

4. Extending the second translation rule to **several** claims and one assumed confidentiality requirement. The claim and the assumed confidentiality requirements concern different variables.

5. Extending the second translation rule to **several** claims and **several** assumed confidentiality requirement. The claim and the assumed confidentiality requirements concern **common variables**.

Translating Minimal Confidentiality Problems into CSP

The minimal non-temporal propositional confidentiality problem has three variables: c_1, a_1 and $\overline{p}$. It is presented below:

claim: full-Confidential(c_1)
assume: full-Confidential(a_1)
behavior: $H_s(c_1, a_1, \overline{p})$
knowledge of unauthorized agents
about the behavior of the system: $H_w(c, a, \overline{p})$

where the variable "c_1" is claimed to be confidential, "a_1" is assumed to be confidential and "$\overline{p}$" is a vector representing the public variables (i.e.: not claimed neither assumed to be confidential).

This restricted problem can be translated into the following equivalent constraint satisfaction problem:

$$\text{find } c_1, a_1, \overline{p}, c' \quad \left(\begin{array}{l} H_s(c_1, a_1, \overline{p}) \\ \\ \wedge\neg(\exists c'_1, a'_1, \overline{p}') \left(\begin{array}{l} H_w(c'_1, a'_1, \overline{p}') \\ \wedge \overline{p}' = \overline{p} \\ \wedge c'_1 = c' \end{array} \right) \end{array} \right)$$

Two constraints must be enforced:

- $H_s(c_1, a_1, \overline{p})$ expresses that the interpretation of c_1, a_1 and $\overline{p}$ must be a model of the system, described by H_s.
- The second constraint expresses that there might be no interpretation $(c'_1, a'_1, \overline{p}')$ that is both admissible in the mind of the attacker (ie: it enforces $H_s(c'_1, a'_1, \overline{p}')$ and the public variables in this interpretation have the same value as in the actual interpretation, that is: $\overline{p}' = \overline{p}$) and such that the claimed confidential variable is equal to some value c'.

This problem can be re-formulated into the equivalent simpler form:

$$\text{find } c_1, a_1, \overline{p}, c' \quad \left(\begin{array}{l} H_s(c_1, a_1, \overline{p}) \\ \wedge(\forall a'_1 \in ran(a_1))\neg H_w(c', a'_1, \overline{p}) \end{array} \right) \tag{4.8}$$

Extending the Simple Translation Rule to One Claim and One Assumed Requirement with a Non-Tautological Premise

We now extend the translation (4.8) in order to support one conditional claim and one conditional assumption (with the pattern $P \rightarrow$ full-Confidential(x)). The assumptions and claims are on distinct variables. This confidentiality problem is defined below:

claim: $P(c_1, a_1, \overline{p}) \rightarrow$ full-Confidential(c_1)
assume: $R(c_1, a_1, \overline{p}) \rightarrow$ full-Confidential(a_1)
assertion: $H_s(c_1, a_1, \overline{p})$
knowledge of unauthorized agents
about the behavior of the system: $H_w(c_1, a_1, \overline{p})$

The conditional claim only adds the constraint that P must also be true in the interpretation that discloses c_1, because

$$\neg(P(c_1, a_1, \overline{p}) \rightarrow \text{full-Confidential}(c_1))$$

is equivalent to

$$P(c_1, a_1, \overline{p}) \wedge \neg\text{full-Confidential}(c_1).$$

The first constraint of (4.8) is strengthened as follows:

$$H_s(c_1, a_1, \overline{p}) \wedge P(c_1, a_1, \overline{p})$$

The conditional assumption is encoded in the problem by modifying the universal quantification. The idea is that if R evaluates to false on the interpretation, we only have to examine the case where $a_1' = a_1$ because the unauthorized agent knows the value of a_1. This is because a is known if not stated to be confidential. Remember that there is at most one claim and one assumption per variable and that our axiomatic system specifies that everything that is not required to be confidential is known to unauthorized agents. However, if R evaluates to true, we have to examine all the possible values for a_1'. This is encoded by modifying the universal quantification as follows:

$$(\forall a_1' \in ran(a_1)) \left(\begin{array}{l} (\neg R(c_1, a_1, \overline{p}) \rightarrow a_1' = a_1) \\ \rightarrow \neg H_w(c', a_1', \overline{p}) \end{array} \right) \tag{4.9}$$

This formula is justified as follows:

- If R evaluates to true, the formula (4.9) reduces to:

  $$(\forall a_1' \in ran(a_1)) \neg H_w(c', a_1', \overline{p})$$

  that is similar to the second constraint of (4.8).

- If R evaluates to false, the formula (4.9) reduces to:

  $$\neg H_w(c', a_1, \overline{p})$$

  Where a_1 is handled similarly to the public variables $\overline{p}$.

The CSP problem with one conditional claim and one conditional assumption is expressed as follows:

$$\begin{array}{l} \text{find } c_1, a_1, c', \overline{p} \\ \left(\begin{array}{l} H_s(c_1, a_1, \overline{p}) \\ \wedge P(c_1, a_1, \overline{p}) \\ \wedge (\forall a_1' \in ran(a_1)) \\ \qquad \left(\begin{array}{l} (\neg R(c_1, a_1, \overline{p}) \rightarrow a_1' = a_1) \\ \rightarrow \neg H_w(c', a_1', \overline{p}) \end{array} \right) \end{array} \right) \end{array} \tag{4.10}$$

Extending the Simple Translation Rule to One Claim and Several Assumed Confidentiality Requirement Conditional Assumptions on Distinct Variables

We consider now that there might be several conditional assumptions. The assumptions and claims are on distinct variables. This confidentiality problem is defined below:

claim: $P(c_1, a_1, a_2, \overline{p}) \rightarrow$ full-Confidential(c_1)
assume: $R(c_1, a_1, a_2, \overline{p}) \rightarrow$ full-Confidential(a_1)
assume: $S(c_1, a_1, a_2, \overline{p}) \rightarrow$ full-Confidential(a_2)
assertion: $H_s(c_1, a_1, a_2, \overline{p})$
knowledge of unauthorized agents
about the behavior of the system: $H_w(c_1, a_1, a_2, \overline{p})$

The corresponding constraint satisfaction problem is (4.11).

$$\begin{array}{l} \text{find } c_1, a_1, a_2, c', \overline{p} \\ \left(\begin{array}{l} H_s(c_1, a_1, a_2, \overline{p}) \\ \wedge P(c_1, a_1, a_2, \overline{p}) \\ \wedge (\forall a_1' \in ran(a_1), a_2' \in ran(a_2)) \\ \qquad \left(\begin{array}{l} (\neg R(c_1, a_1, a_2, \overline{p}) \rightarrow a_1' = a_1) \\ \wedge (\neg S(c_1, a_1, a_2, \overline{p}) \rightarrow a_2' = a_2) \\ \rightarrow \neg H_w(c', a_1', a_2', \overline{p}) \end{array} \right) \end{array} \right) \end{array} \tag{4.11}$$

This is a systematization of (4.10). For instance, if $R(c_1, a_1, a_2, \overline{p})$ evaluates to false and $S(c_1, a_1, a_2, \overline{p})$ evaluates to true, (4.11) reduces to the following:

$$\begin{array}{l} \text{find } c_1, a_1, a_2, c', \overline{p} \\ \left(\begin{array}{l} H_s(c_1, a_1, a_2, \overline{p}) \\ \wedge P(c_1, a_1, a_2, \overline{p}) \\ \wedge (\forall a_1' \in ran(a_1)) \\ \qquad \left(\ \neg H_w(c', a_1', a_2, \overline{p}) \ \right) \end{array} \right) \end{array}$$

This reduced formula is similar to (4.8).

Extending the Simple Translation Rule to Several Conditional Claims and One Conditional Assumption on Distinct Variables

We consider now that there might be several conditional claims. For clarity, we only consider one conditional assumption. The assumptions and

claims are on distinct variables. This confidentiality problem is defined below:

claim: $P(c_1, c_2, a_1, \overline{p}) \rightarrow$ full-Confidential(c_1)
claim: $Q(c_1, c_2, a_1, \overline{p}) \rightarrow$ full-Confidential(c_2)
assume: $R(c_1, c_2, a_1, \overline{p}) \rightarrow$ full-Confidential(a_1)
assertion: $H_s(c_1, c_2, a_1, \overline{p})$
knowledge of unauthorized agents
about the behavior of the system: $H_w(c_1, c_2, a_1, \overline{p})$

As there are several claimed-confidential variables, we add a new variable "d" that represents the disclosed variable. "d" takes the values 1 if the disclosed variable is c_1 and 2 if the disclosed variable is c_2. The corresponding constraint satisfaction problem is (4.12).

$$\begin{array}{l}\text{find } c_1, c_2, a_1, d, c', \overline{p} \\ \left(\begin{array}{l} H_s(c_1, c_2, a_1, \overline{p}) \\ \wedge d = 1 \rightarrow P(c_1, c_2, a_1, \overline{p}) \\ \wedge d = 2 \rightarrow Q(c_1, c_2, a_1, \overline{p}) \\ \wedge (\forall a_1' \in ran(a_1), c_1' \in ran(c_1), c_2' \in ran(c_2)) \\ \qquad \left(\begin{array}{l} (\neg R(c_1, c_2, a_1, \overline{p}) \rightarrow a_1' = a_1) \\ \wedge d = 1 \rightarrow c_1' = c' \\ \wedge d = 2 \rightarrow c_2' = c' \\ \rightarrow \neg H_w(c_1', c_2', a_1', \overline{p}) \end{array} \right) \end{array} \right) \end{array} \tag{4.12}$$

The value c' is included in $ran(c_1) \cup ran(c_2)$. For instance, if $d = 1$, it means that c_1 is disclosed. In this case, (4.12) reduces to the following: We might have split the problem into different CSPU. This would have prevented us from introducing the variable c'. Here, we preferred to keep one single problem because we felt that these different problems share a very large common search space.

$$\begin{array}{l}\text{find } c_1, c_2, a_1, c', \overline{p} \\ \left(\begin{array}{l} H_s(c_1, c_2, a_1, \overline{p}) \\ \wedge P(c_1, c_2, a_1, \overline{p}) \\ \wedge (\forall a_1' \in ran(a_1), c_2' \in ran(c_2)) \\ \qquad \left(\begin{array}{l} (\neg R(c_1, c_2, a_1, \overline{p}) \rightarrow a_1' = a_1) \\ \rightarrow \neg H_w(c', c_2', a_1', \overline{p}) \end{array} \right) \end{array} \right) \end{array}$$

This reduced formula is similar to (4.10), with c_2 taken as unconditionally assumed confidential variable (that is: c_2' appears in the universal

quantification).

Extending the Simple Translation Rule to Several Conditional Claims and Several Conditional Assumptions With Common Variables

We present now the translation for general confidentiality problems. General confidentiality problems are actually made of a list of conditional confidentiality claims and assumptions. For instance, the running example matches the pattern:

claim: $P(c_1, c_2, \overline{p}) \rightarrow$ full-Confidential(c_1)
claim: $Q(c_1, c_2, \overline{p}) \rightarrow$ full-Confidential(c_2)
assume: $R(c_1, c_2, \overline{p}) \rightarrow$ full-Confidential(c_1)
assume: $S(c_1, c_2, \overline{p}) \rightarrow$ full-Confidential(c_2)
assertion: $H_s(c_1, c_2, \overline{p})$
knowledge of unauthorized agaents
about the behavior of the system: $H_w(c_1, c_2, \overline{p})$

The difference with the previous problem is that the claimed confidential variables are also assumed to be confidential. In this situation, while violating the claim on a variable, we do not consider the assumed confidentiality requirements about this variable. By using the mechanisms presented above, including the "d" variables that captures the identity of the disclosed variable, we can express the general confidentiality problem as a CSP in (4.13). The variable d takes the values 1 if the disclosed variable is c_1 and 2 if the disclosed variable is c_2.

$$\begin{array}{l}\text{find } c_1, c_2, d, c', \overline{p} \\ \left(\begin{array}{l} H_s(c_1, c_2, \overline{p}) \\ \wedge d = 1 \rightarrow P(c_1, c_2, \overline{p}) \\ \wedge d = 2 \rightarrow Q(c_1, c_2, \overline{p}) \\ \wedge (\forall c_1' \in ran(c_1), c_2' \in ran(c_2)) \\ \qquad \left(\begin{array}{l} d = 1 \rightarrow c_1' = c' \\ \wedge d \neq 1 \rightarrow (\neg R(c_1, c_2, \overline{p}) \rightarrow c_1' = c_1) \\ \wedge d = 2 \rightarrow c_2' = c' \\ \wedge d \neq 2 \rightarrow (\neg S(c_1, c_2, \overline{p}) \rightarrow c_2' = c_2) \\ \rightarrow \neg H_w(c_1', c_2', \overline{p}) \end{array} \right) \end{array} \right) \end{array} \quad (4.13)$$

For instance, if we consider the case that $d = 1$, it means that c_1 is disclosed. Therefore, the input problem can be reduced into:

claim: $P(c_1, c_2, \overline{p}) \rightarrow$ full-Confidential(c_1)

assume: $S(c_1, c_2, \overline{p}) \rightarrow$ full-Confidential(c_2)
assertion: $H_s(c_1, c_2, \overline{p})$
knowledge of unauthorized agents
about the behavior of the system: $H_w(c_1, c_2, \overline{p})$

and the formula (4.13) reduces to:

$$\text{find } c_1, c_2, c', \overline{p} \quad \left(\begin{array}{l} H_s(c_1, c_2, \overline{p}) \\ \wedge P(c_1, c_2, \overline{p}) \\ \wedge (\forall c_2' \in ran(c_2)) \\ \qquad \left(\begin{array}{l} (\neg S(c_1, c_2, \overline{p}) \rightarrow c_2' = c_2) \\ \rightarrow \neg H_w(c', c_2', \overline{p}) \end{array} \right) \end{array} \right)$$

By identification with (4.10), one can notice that the reduced constraint satisfaction problem solves the reduced input problem.

4.5.2.6 Solving Constraint Satisfaction Problems

The constraint satisfaction problem generated by the previous layer (4.13) has the following pattern:

$$\text{find } \overline{x} \in \overline{X} \quad \left(\begin{array}{l} P(\overline{x}) \\ \wedge (\forall \overline{y} \in \overline{Y}) Q(\overline{x}, \overline{y}) \end{array} \right) \tag{4.14}$$

Where $\overline{x}$ and $\overline{y}$ are vectors of domain $\overline{X}$ and $\overline{Y}$ respectively. These domains are finite sets. $P(\overline{x})$ and $Q(\overline{x}, \overline{y})$ represent constraints on $\overline{x}$ and $(\overline{x}, \overline{y})$ respectively. We call such problem CSPU for "Constraint Satisfaction Problem with a Universal quantification".

Such constraint satisfaction problems require specific techniques that can be implemented on top of traditional constraint solvers. We have proposed a specialized extension of traditional constraint programming techniques to handle such particular problems [Lan04].

Existing tools have become available to solve such problems. These are *QBF solvers* [RN03]. QBF stands for *quantified boolean formulas*. We did not use them because QBF solvers do not support integers and our input problem includes the notion of integers and might include operators such as addition and comparison. Integers are efficiently managed by constraint programming techniques.

The difficulty of solving this problem through standard constraint programming technique is that we need to post constraints at the beginning of the search. The constraint Q is nested in a universal quantification. We therefore need to post a constraint of each of the value of Y. This posting gives rise to a problem with a lot of constraints, which consumes a prohibitive amount of memory. The solution was to add the Q constraints "on demand".

We present below our basic principle for solving this problem. The explanation presented below is based on a naive approach that is criticized and leads to the elaboration of a better one. The implemented method is an optimization of the presented one and can be found in [Lan04].

In order to avoid heavy notations, the approach is presented on single values instead of vectors. The generalization is straightforward.

Note that the implementation requests particular attention because this layer is the time critical layer. Therefore, the implementation uses some advanced techniques such as recomputation, adaptative recomputation, speculative exploration of solution space and abstraction.

The naive approach to solve CSPU consists in solving the expanded problem where y has been instantiated to each of the possible values in Y. The expanded problem is presented in (4.15) where $y_1, y_2, ...$ are the values in Y [1].

$$(\exists x \in X) \left(\begin{array}{l} P(x) \\ \wedge Q(x, y_1) \wedge Q(x, y_2) \wedge Q(x, y_3) \wedge ... \end{array} \right) \tag{4.15}$$

This approach has many deficiencies:

- it is time consuming because a large number of $Q(x, y_i)$ must be generated by the expansion procedure, as the cardinality of Y might be very large;
- it is time consuming because all the $Q(x, y_i)$ must be handled by the search procedure, i.e: cloned when the computation space is cloned by the search engine;
- it is memory consuming because all the $Q(x, y_i)$ must be stored in memory.

This algorithm aborts for simple problems because of excessive memory consumption. A classical method used in constraint programming that is *recomputation* makes it better, but still unusable.

[1] In this section, the convention is that subscripted letters are values and non subscripted letters are variables. Moreover, x (resp y) denotes a variable whose domain is X (resp Y). For instance, y_1, y_2 and y_i are values in Y and x_1, x_2 and x_j are values in X

The problems of the naive approach appear because the problem is very heavy, because it contains too many constraints. Our objective is to make this problem lighter in order to make it possible to perform search with it.

The basic observation is that not every $Q(x, y_i)$ is useful. Many of these $Q(x, y_i)$ are either redundant or not exploited by the search engine to find the *first* solution. This leads to the idea that we should make the problem lighter for the search engine in order to achieve better efficiency, by injecting only the useful $Q(x, y_i)$. In order to express this idea and to reason on it, we introduce here the concept of representative constraint.

A *representative* constraint is one such that some other constraints are redundant to it. A *good* representative constraint is one such that *many* constraints are redundant to it. The other constraints can be removed from the system without changing the result. In our case, we extend the notion of representative constraint to representative *value* because the $Q(x, y_i)$ are determined by the y_i. This concept is similar to the one used in black box software test case generation [Bor95]. The concept of representative value can be formalized in our context as follows: *A value y_i is a good representative if there are many $y_j \in Y$ such that for all $x_k \in X$: $Q(x_k, y_i) \rightarrow Q(x_k, y_j)$.*

The basic idea is to solve a simplified problem with only good representative $Q(x, y_i)$. However, we don't know which are the good representative $Q(x, y_i)$. The solution presented here consists in finding the representative constraint *during* the search. Our approach is then highly dependent on the existence of such representatives. We elaborate on this idea in the next section.

We first solve the simplified problem

$$(\exists x \in X)P(x)$$

If no solution is found to this simplified problem, no solution for the complete problem exists either. If a solution x_1 is found, we search a solution to the auxiliary problem

$$(\exists y \in Y)\neg Q(x_1, y)$$

- If a solution y_1 is found to the auxiliary problem, then x_1 was not a good solution. We make the assumption that y_1 is a good representative. We strengthen the initial simplified problem with $Q(x, y_1)$ and restart the algorithm.
- If no solution is found to the auxiliary problem, then x_1 is a solution to the initial problem as $\neg\exists y\neg Q(x_1, y)$ is equivalent to $\forall y Q(x_1, y)$

The initial constraint $P(x)$ is gradually strengthened with some $Q(x, y_i)$ until either the problem cannot be solved or a solution to the CSPU problem

is found. We hope that the strengthened problem will be much lighter than the expanded CSPU problem of the naive approach. The search of the form $(\exists x \in X)P(x)$ is called the *main* search while the search of the form $(\exists y \in Y)\neg Q(x_1, y)$ is called the *auxiliary* search. A pseudo-code formulation of the algorithm is given in Fig. 1.

Algorithm 1 Search_With_Universal_Quantification(P, X, Q, Y)

```
% This function returns a solution to the problem:
% (∃x ∈ X)(P(x) ∧ (∀y ∈ Y)Q(x, y)) Where X and Y are finite sets.
% It returns no_solution if no solution exists.

% Main search
Search for a solution to (∃x ∈ X)P(x)
if no solution is found then
  % There is no solution to (∃x ∈ X)P(x),
  % thus no solution to the original problem
  return no_solution
else if there is a solution x1 then
  % Here, we know that P(x1) but, we don't know if ∀yQ(x1, y)
  % Auxiliary search
  Search for a solution to (∃y ∈ Y)¬Q(x1, y)
  if no solution is found then
    % x1 is a solution to the original problem,
    % as ¬∃y¬Q(x1, y) ≡ ∀yQ(x1, y)
    return x1
  else if there is a solution y1 then
    % x1 was not a solution to the original problem
    %as ¬Q(x1, y1) holds.
    % Strengthen P(x) with Q(x, y1) and solve the new problem.
    let R = (λx)(P(x) ∧ Q(x, y1))
    return Search_With_Universal_Quantification(R, X, Q, Y)
  end if
end if
end function
```

The implementation of the CSPU solver algorithm is based on a branch and bound algorithm that behaves similarly to the algorithm presented in Figure 1. Moreover, we use a specialized propagator that imperfectly captures the universal quantification in order to improve the search. These technicalities are presented in [Lan04].

4.5.2.7 Producing Attacker's Reasoning as an Auxiliary Output

The trace T and value c discovered by CONCHITA has the property that there is no trace T' such that:

$$\begin{aligned} & T' \models H_w \\ & \wedge \text{non-confidential variables have same values in } T \text{ and } T' \\ & \wedge \text{claimed-confidential variable in } T' \text{ is equal to some value } c' \end{aligned} \tag{4.16}$$

The meaning of this non-existence of T' is that given the knowledge of H_w, and his knowledge of the value of non-confidential variables, the unauthorized agent is able to distinguish the real trace T from all traces T' where the claimed confidential variable has the value c. The attacker's knowledge of the system is actually a conjunction of knowledge fragments: $H_w = \wedge_i H_{w,i}$. For instance, each requirement might be considered as a (coarse-grained) fragment of knowledge. We want to know which of the fragments $H_{w,i}$ is necessary to the unauthorized agent to get to this conclusion. This problem can be defined in term of conflict set and minimal conflict set:

Definition (*Conflict set*): Given a constraint satisfaction problem of the form "find $x | \wedge_i P_i(x)$" that has no solution, a conflict set C is a subset of $\{P_i\}_i$'s such that the constraint satisfaction problem "find $x | \wedge_{i \in C} P_i(x)$" has no solution.

Definition (*Minimal conflict set*): A conflict set is minimal if removing any of its element causes the related problem to have a solution.

Minimal conflict sets can be computed efficiently by standard algorithms such as QuickXplain [Jun01].

Let $\{H_{w,i}\}_i$ be the fragments such that $H_w = \wedge_i H_{w,i}$. The minimal set of fragments that are necessary to ensure that the problem (4.16) where T' is to be determined has no solution is a "minimal conflict set" for this problem and can be solved by standard algorithms.

CONCHITA therefore generates fragments of H_w and computes the minimal conflict set on these fragments through the QuickXplain algorithm [Jun01]. In order to characterize the unauthorized agent's reasoning as precisely as possible, the fragment of knowledge are made as fine-grained as possible.

Fragmentation is performed by the propozitionalization and the time bounding layers. For example, a universal quantification yields one fragment per instance, a "always" operator yields one fragment per position between the position at which it is evaluated and the last position in the trace, and a conjunction yields two fragments. Fragmentation is performed recursively

up to the first operator that cannot be fragmented; generally an implication or a disjunction.

4.5.3 Properties of CONCHITA

Our checker is sound; if it finds a trace, the trace does violate some confidentiality claim while matching the system specification. In other words, it will never produce false positives.

Like any bounded model checker, CONCHITA only searches within the given time bounds and object instances. Assigning too restrictive bounds can therefore result in failure to find a trace when there is a longer one or one with more instances.

We do believe that CONCHITA is bounded-complete: if there is a trace within the given bounds, it will find it. Bounded-completeness is achieved by ensuring that the time-bounding and propositionalization layers perform as little strengthening/weakening as possible. Completeness issues do not arise at the two lower layers which perform exact translations and resolutions of their input problem.

The complexity of CONCHITA is however very bad. Solving a constraint satisfaction problem including a universal quantification is similar to solving a quantified boolean formula problem (QBF). Testing that a quantified boolean formula is true or false is PSPACE-COMPLETE and NP-COMPLETE in the number of variables, supposed boolean, appearing in the quantifiers [Sip06, Pap94]. Nevertheless, the use of constraint programming as a search engine enables us to use various heuristics for exploring promising solutions first. These heuristics generally keep the computing time low for the examples discussed here. Developing an efficient search algorithm for solving confidentiality-checking problems is however a very important direction for future work.

4.6 Managing Confidentiality Requirements With CONCHITA

This section presents how confidentiality requirements are analyzed at requirements engineering time with help of CONCHITA. Additional security requirements are derived to ensure that the confidentiality requirements are enforced by the requirements model.

We first present a simple process for enforcing confidentiality. We illustrate it on the e-signature case study. We finally discuss an artifact of confidentiality analysis within declarative framework.

4.6.1 A Confidentiality Enforcement Process and Tactics

Figure 4.11 presents a process for ensuring that confidentiality requirements are enforced by a requirements model. The steps in Figure 4.11 are ordered by data dependency and generally intertwined.

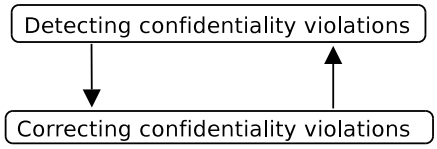

Figure 4.11: Confidentiality requirement enforcement process

The two steps presented in Figure 4.11 are:

- Detecting confidentiality requirement violations. This is performed by applying the CONCHITA tool on the fragmentary requirements model.
- Resolving detected confidentiality violation. This is performed by applying one of the following tactics to the detected confidentiality violation:
  - **Specifying additional confidentiality requirements** on variables that leak information to unauthorized recipients. This tactic can only be applied if it is possible to enforce these new confidentiality requirements.
  - **Tolerate the disclosure** and weaken the confidentiality goals accordingly. This is performed either by decreasing the set of unauthorized agents, or by lowering the specified timing or degree of confidentiality.
  - **Restricting functionalities** causing the confidential data to be disclosed. Confidentiality violations are due to functionalities of the system-to-be. It might be the case that it refused to do something, as in the case of the e-purse presented in Section 4.5.1 or some requirement might prescribe that the value of the confidential variable must be propagated to some other non-confidential variable.

  Choosing between the three tactics must be performed according to the needs expressed by the stakeholders or the domain experts.

For instance, the confidentiality violation identified in the e-purse in Section 4.5.1 can be resolved by weakening the confidentiality requirement from full-Confidential-forever to lb-Confidential-now.

In the next subsection, we illustrate this process on the electronic signature system. We also illustrate the tree tactics for resolving confidentiality violations All input and output of CONCHITA discussed in this section are reported in Annex A.4.

4.6.2 Illustrating the Process and Tactics

Illustrating the Process and Tactics on the e-signature System

We examine the confidentiality goal *Maintain[Signer Password confidentiality]*. This goal is fully defined as follows:

Goal Maintain[Signer Password confidentiality]
Definition The password stored in an E-ID should only be known to the E-ID itself, to the Signer represented by this E-ID and to genuine Sign Boxes. This is full instantaneous confidentiality
Formal Def($\forall e$: EID a: Agent)
$\neg$Represent<eid: e signer: a>
$\wedge\neg a = e$
$\wedge\neg$Genuine<s: a>
$\Rightarrow$ full-Confidential-now$_a$(e.password)

We first examine the confidentiality breach due to the functional goal *Achieve[password entered and E-ID inserted into genuine sign box when agreement on displayed document]*. this goal is fully defined as follows:

Goal achieve[password entered and E-ID inserted into sign box when agreement on displayed document]
Definition When a signer is willing to sign a document displayed by a SignBox, he shoud insert his E-ID in it and enter his password into it.
Formal Def($\forall s$: SignBox e: EID i: Signer)
AgreesOnDisplayedFile<who: i where: s>
$\wedge$Inserted<what: e into: s>
$\wedge$Represent<eid: e signer: i>
$\wedge$at<who: i where: s>
$\Rightarrow \circ$(EnteredPassword<into: s val: e.password>
$\wedge$Inserted<what: e into: s>
$\wedge$at<who: i where: s>)

In order to test the fragment of requirements model for confidentiality violations, we have produced the first input file presented in Annex A.4. It includes the object model presented in Figure 4.12, the two goals defined above and two auxiliary domain properties.

When submitted with this confidentiality test problem, CONCHITA delivers the following trace where the password is disclosed to any external

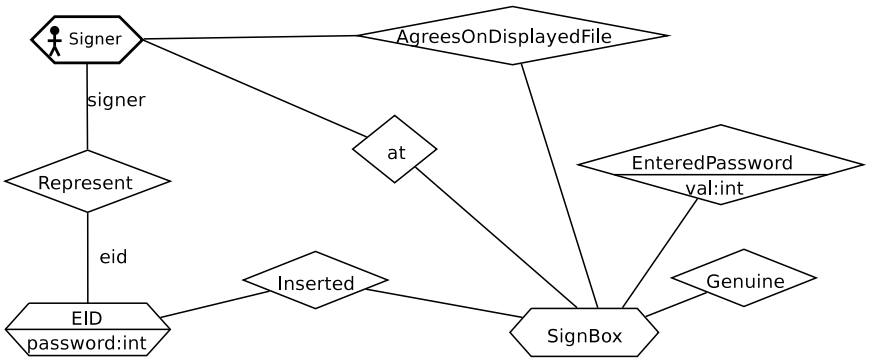

Figure 4.12: Object model of the e-signature system

agent (see first input file and trace in Annex A.4):

```
---------------------------- State 0
AgreesOnDisplayedFile<
        where:SignBox0
        who:bobby>
EID0  (EID agent){
        password:0}
Inserted<
        into:SignBox0
        what:EID0>
Genuine<
        s:SignBox0>
Represent<
        eid:EID0
        signer:bobby>
at<
        where:SignBox0
        who:bobby>
---------------------------- State 1
EID0  (EID agent){
        *password:0*     (DISCLOSED at the end of the trace)}
EnteredPassword<
        into:SignBox0
        val:0>
Inserted<
        into:SignBox0
        what:EID0>
Genuine<
        s:SignBox0>
```

```
Represent<
        eid:EID0
        signer:bobby>
at<
        where:SignBox0
        who:bobby>

At this state, any external agent knows that
EID0.password is different from 1 at time 1
----------------------------
```

In this trace, the external agent can get to know the password of E-ID0 because bob, the owner of EID0 enters it into the SignBox0. This is denoted by the association `EnteredPassword<into:SignBox0 ...>` at time 1. The dialed value is not confidential and can therefore be observed by any external agent.

We prevent this disclosure by adding the confidentiality goal *maintain[Entred Password Confidential]*. This goal is fully defined as follows:

Goal maintain[entered password confidential]
Definition The password entered into a SignBox should only be known to the SignBox itself and to the Signer that is at the SignBox
Formal Def($\forall s$: SignBox a: Agent)
$\neg a = s$
$\wedge \neg$at<who: a where: s>
$\Rightarrow$ full-Confidential-now$_a$(EnteredPassword<into: s>.val)

This is a first tactics for correcting detected confidentiality violations:

Tactics 1 *extending confidentiality requirement to leaked value*

The requirements model increased with this confidentiality goal is submitted to CONCHITA (see second input file and trace in Annex A.4). CONCHITA delivers a trace leading to a disclosure of the entered password. This corresponds to a violation of our new confidentiality goal. The trace is as follows:

```
---------------------------- State 0
AgreesOnDisplayedFile<
        where:SignBox0
        who:bobby>
EID0  (EID agent){
        password:0}
```

```
Inserted<
        into:SignBox0
        what:EID0>
Genuine<
        s:SignBox0>
Represent<
        eid:EID0
        signer:bobby>
at<
        where:SignBox0
        who:bobby>
---------------------------- State 1
EID0  (EID agent){
        password:0}
EnteredPassword<
        into:SignBox0
        *val:0*  (DISCLOSED at the end of the trace)>
Inserted<
        into:SignBox0
        what:EID0>
Genuine<
        s:SignBox0>
Represent<
        eid:EID0
        signer:bobby>
at<
        where:SignBox0
        who:bobby>

At this state, EID0 knows that
field val of instance of EnteredPassword
is different from 1 at time 1
----------------------------
```

This trace shows that the EID can get to know the password entered into the SignBox it is inserted in because this password is equal to the one stored in itself. This is a spurious confidentiality violation arising because the goal *Maintain[entered password confidential]* is too strong. An E-ID should be allowed to know the password entered into the SignBox it is inserted in.

This confidentiality violation is resolved by weakening the confidentiality goal so that agents with a need to know were listed as authorized ones. In this situation, the E-ID inserted into a SignBox is allowed to know the password entered in it.

This is a tactics for resolving confidentiality violation:

Tactics 2 *weakening confidentiality goals by extending the set of authorized agents*

The weakened confidentiality goal is fully defined as follows:

Goal maintain[entered password confidential]
Definition The password entered into a SignBox should only be known to the SignBox itself and to the Signer that is at the SignBox
Formal Def($\forall s$: SignBox a: Agent)
$\neg a = s$
$\wedge\neg$at<who: a where: s>
$\wedge\neg$Inserted<what: $a$into: s>
$\Rightarrow$ full-Confidential-now$_a$(EnteredPassword<into: s>.val)

The resulting requirements model is again submitted to CONCHITA (see third input file and trace in Annex A.4). CONCHITA discovers another confidentiality violation. The trace is as follows:

```
---------------------- State 0
AgreesOnDisplayedFile<
        where:SignBox0
        who:bobby>
EID0  (EID agent){
        password:0}
Inserted<
        into:SignBox0
        what:EID0>
Represent<
        eid:EID0
        signer:bobby>
at<
        where:SignBox0
        who:bobby>
--------------------------- State 1
EID0  (EID agent){
        *password:0*     (DISCLOSED at the end of the trace)}
EnteredPassword<
        into:SignBox0
        val:0>
Inserted<
        into:SignBox0
        what:EID0>
```

```
Represent<
        eid:EID0
        signer:bobby>
at<
        where:SignBox0
        who:bobby>

At this state, SignBox0 knows that
EID0.password is different from 1 at time 1
----------------------------
```

In this trace, the EID has been inserted into Signox0 and the password entered into it. The point is that SignBox0 is not a genuine one and as such, it is not allowed to know the password of EID0.

In order to prevent this confidentiality violation, we add a security goal stating that the password should not be entered into a non-genuine SingBox.

This is a tactics for resolving confidentiality violation:

Tactics 3 *restricting functionalities causing information flow*

The weakened functional goal is fully defined as follows:

Goal avoid[password entered into non genuine sign box]
Definition The password of an EID should not be entered into a SignBox that is not a genuine one
Formal Def$(\forall s\colon \text{SignBox})$
$\neg\text{Genuine}\langle s\colon s\rangle$
$\Rightarrow \neg\text{EnteredPassword}\langle\text{into}\colon s\rangle$

This new goal conflicts with the functional goal *Achieve[password entered into sign box when agreement on displayed document]*. We therefore have to weaken the latter by stating that the SignBox in which the password is entered and the E-ID is inserted must be a genuine one. The new definition is as follows:

Goal achieve[password entered and E-ID inserted into genuine sign box when agreement on displayed document]

Definition When a signer is willing to sign a document displayed by a genuine SignBox, he should insert his E-ID in it and enter his password into it.

Formal Def$(\forall s\colon \text{SignBox}\ e\colon \text{EID}\ i\colon \text{Signer})$
AgreesOnDisplayedFile<who: i where: s>
$\wedge$Inserted<what: e into: s>
$\wedge$Represent<eid: e signer: i>
$\wedge$Genuine<s: s>
$\wedge$at<who: i where: s>
$\Rightarrow \circ$(EnteredPassword<into: s val: e.password>
$\wedge$Inserted<what: e into: s>
$\wedge$at<who: i where: s>)

When fed with this new requirements model fragment, CONCHITA does not find confidentiality violation anymore (see fourth input file and output of CONCHITA in Annex A.4). We tried to increase the length of the trace to three, but this did not enable CONCHITA to find a confidentiality violation.

In order to ensure that the confidentiality goals are enforced, we must include the other goals of the system to ensure that they do not disclose confidential information. Mainly, we need to examine the functional goal *achieve[entered password forwarded to inserted eid]*.

4.6.3 Confidentiality and Declarativeness

In our example, the last round of our confidentiality enforcement process illustrates an issue of our framework. It is due to the declarativeness and non-monotonicity of our framework (see Section 4.3.2).

There was a confidentiality violation because users are required to enter their password into a SignBox that might not be a genuine one. We resolved it by adding a security goal prescribing users not to enter their password into non-genuine SignBoxes. We then noticed that this new goal was conflicting with the requirement prescribing that they should enter it into a SignBox when they are willing to sign an electronic document. These goals were inconsistent because they might be willing to sign a document while sitting at a non-genuine signbox. The functional goal was weakened accordingly, to resolve the conflict.

The point is that this weakening alone is enough to remove the confidentiality violation. We have produced an input file for CONCHITA where the functional goal is weakened. This file does not include the security goal (see fifth input file and output of CONCHITA in Annex A.4). CONCHITA does

not find any confidentiality violation. This is not because CONCHITA is not complete. This is because if a password is entered into a SignBox that is not a genuine one, the attacker modeled within CONCHITA does not have enough knowledge to conclude that the entered value is the one associated with a given E-ID.

In this model, there is no security goal explicitly prescribing that users should not enter their password in non-genuine SignBoxes, yet there is no confidentiality violation detected. Because of this, we might fail to find some necessary security goal prescribing that they should not enter their password into non-genuine SignBoxes. More generally, important security subgoals might be unidentified. For instance, we might think of a mechanism enabling users to spot non-genuine SignBoxes.

The declarativeness inherent to our goal-oriented approach might hinder our capabilities of detecting confidentiality violations at early time of requirements modeling. This is a well-known issue in confidentiality modeling [Jac89]. Of course, the potentially undetected confidentiality violations can be detected once an executable model is derived from our declarative requirements. This derivation process is known to lift non-determinism that might appear in the requirements [REA04]. This might induce additional confidentiality violations [Jac89].

4.7 Related Work

Confidentiality analysis has been carried out on transition systems and process algebra representing program behaviors.

[SR05] presents an approach for capturing information flow in transition systems and enables confidentiality analysis. This approach requires that an operational description of the system-to-be is available and makes the assumption that confidentiality violation always occur because of a direct confidentiality violation, that is: classified information is directly displayed to some unauthorized agent. It is particularly well suited for analyzing confidentiality requirements on program code or security policies.

[BDR04] goes one step further and also includes the notion of indirect confidentiality violation where the unauthorized agent gets to know classified information by performing deductive reasoning, similarly to the approach presented here. It also shows that an extended logic is needed to capture indirect confidentiality violation.

In contrast with these two approaches, our technique is suited for reasoning in a declarative way on partial requirements models. We also provide a richer set of confidentiality patterns, that includes temporal referencing.

It is worth noting that within these frameworks, it is possible to prove that a confidentiality goal is not violated because the description is operational and supposed complete. A confidentiality proof produced within these frameworks is however sound up to the confidentiality leaks that might be introduced during the implementation [Jac89]. This is due to the very nature of confidentiality.

4.8 Conclusion

Confidentiality is an important class of security concerns that refers to restrictions on agent knowledge. The earliest security leaks are detected, the best; countermeasures can then be devised early as new requirements for a more robust system. Our formal framework makes it possible to specify and analyze confidentiality concerns at requirements engineering time. Non-intentional and malicious disclosure of sensitive information are both supported. Our framework has three components.

- An axiomatic system, built on top of epistemic logics [FHMV95], captures how the knowledge of unauthorized agents evolves over time. In a way similar to the "Most Powerful Attacker" hypothesis used in cryptographic protocol analysis, we assume unauthorized agents to be clever and "forgetless".
- A catalog of confidentiality specification patterns captures different types of confidentiality requirements on state variables. The patterns are expressed in terms of knowledge operators from this axiomatic system. The catalog is built on top of previous patterns [vL04]. It has a two-dimensional structure according to the degree of approximate knowledge and knowledge timing.
- A proof-of-concept tool prototype checks requirements models against confidentiality requirements expressed in terms of such epistemic patterns. The tool separately maintains unauthorized agents' knowledge according to the epistemic axiomatic system. It combines bounded model checking and constraint programming techniques to generate counter-example traces leading to violation of confidentiality requirements.

Our framework combines formality, grounded on epistemic logics, and lightweightedness through specification patterns.

Formality is a prerequisite for a well-defined semantics of specification constructs and for in-depth analysis of interactions between functional and security requirements. The analysis performed by the tool is incremental as

it is applied to declarative model fragments. It is made efficient by instantiation of our models to small worlds, as done by bounded model checkers, and by pruning of the search space, as done by constraint solvers. The latter more efficiently supports complex specifications, for example, over arithmetic expressions.

The patterns, in the spirit of [DAC98, CP99], considerably facilitate the task of writing specifications with knowledge operators. They can also be used as a means for eliciting confidentiality requirements and making them more precise.

Our framework makes minimal assumptions about the requirements model. It essentially needs a simple form of UML class diagram and a set of assertions. It might therefore be grafted at low cost on recent extensions of goal-oriented and agent oriented methods for threat modeling [LYM03, vL04]. Moreover, CONCHITA can be used in the context of goal-oriented requirements engineering [vL01] to ensure that some high-level goal is enforced by lower-level requirements. In a banking application, for example, we should ensure that bank accounts are confidential on the basis that information flows between ATMs and the central bank is kept confidential; in such a case the high-level goal would be submitted as a confidentiality claim whereas the lower-level requirements are submitted as assumed confidentiality requirements.

4.9 Open Issues

In situations where confidentiality violations are not felt critical, it might be worth considering less clever unauthorized agents who know less about the system or are forgetful. In such situations we should then explicitly provide application-specific axioms that specify the conditions under which specific system knowledge fragments appear or disappear from the memory of unauthorized agents.

Another concern about the attacker model is that it only includes deductive reasoning. In real world, attackers might also use inductive reasoning in order to get additional knowledge, which is then regarded as a belief as it might be false. Such reasoning might also be considered, but it is not clear whether it will enable the discovery of vulnerabilities in a requirements model that cannot be discovered by a fully deductive agent.

Our approach focuses on the knowledge of a single agent. Attacks on confidentiality goals might require the collusion of several attackers and the sharing of information. An interesting open issue might consist in extending our approach and our technique in order to consider such collusions. Typically, our confidentiality checking technique would search for a set of attacker

agents who (1) get enough knowledge for violating the confidentiality claim, through information sharing and (2) such that none of the attacker agents is authorized for the confidentiality claim.

Our approach requires the analyst to specify all necessary confidentiality requirements (in view of the maximal input axiom). We might alleviate this by imposing a least privilege principle [Kem03] as additional requirement on some agents, for example, software agents (see Section 2.3.4.3). This principle would state here that the agents may only see those state variables which they need to monitor/control for achieving the requirements they are responsible for.

Our pattern catalog is currently limited to confidentiality of attributes of entity. It can be extended to capture the confidentiality of associations and of attributes of associations in order to provide more expressiveness when specifying confidentiality requirements. Our pattern library exclusively focusses on specifying that a single variable should be confidential. There might be interesting extensions to our patterns related to the confidentiality of facts involving several variables.

Although it was built in a systematic way along two structuring dimensions, its coverage needs to be better assessed through further validation studies. Our current implementation also needs to be extended to support the entire catalog, and then experimented on larger-scale applications.

Bounded-model-checking is a known efficient technique for solving temporal problems, thanks to recent advances in SAT-solving and constraint programming techniques. Despite the attractiveness of efficiency, it remains bounded in that it might fail to disclose some confidentiality violations because the bounds given to it are too coarse. It might be interesting to look for "unbounded" model-checking techniques for solving epistemic-temporal problems. Recent advances in symbolic manipulation techniques such as the antichain lattice data structure for efficiently representing sets of sets of states might make it possible to produce a viable tool for epistemic and temporal model-checking [WDR06].

Chapter 5

Assigning Responsibilities for Security Requirements

Some agents might be malevolent or subject to threats. Making them responsible for security requirements or assumptions may introduce vulnerabilities that might result in security violations. This is the issue we address in this chapter.

This chapter proposes an extension of the goal-oriented approach presented in Chapter 3. This extension ensures that security goals are enforced even in the presence of malevolent agents. The principle is to ensure that the requirements assigned to trustworthy agents together enforce the high-level security goals. Untrustworthy agents include malevolent or potentially malevolent agents. We argue that untrustworthy agents are easier to spot when reasoning about high-level security goals than when reasoning about requirements. Our extension builds on a dedicated model of trustworthiness. This chapter proposes a security-specific completeness criterion for requirements models. Our criterion builds on this concept of trustworthiness.

The chapter is organized as follows: Section 5.1 presents why responsibility assignments are crucial and introduces the concept of trustworthy agent, Section 5.2 introduces a goal-oriented notion of agent trustworthiness and formally characterizes a criterion for requirements completeness on the basis of this concept. Section 5.3 discusses countermeasures and axioms they must satisfy for achieving our requirements completeness criterion. Section 5.4 details a process for building goal models that enforces our requirements completeness criterion. Section 5.5 discusses the use of formal analysis tools for supporting the process. Section 5.6 presents an evaluation of our technique on the e-purse system. We compared our requirements model of the e-Purse with a list of requirements issued by the European Central Bank on smartcard-based e-purse systems. Section 5.7 discusses some related work. Section 5.9 presents some open issues.

5.1 Overview

Preliminary principles and heuristics for responsibility assignment have been proposed in various RE frameworks.

- In KAOS, a human agent should not be made responsible for a goal conflicting with its own personal interests [DvLF93]. Techniques for detecting conflicts of interests have been proposed in [GMMZ06].
- In the spirit of the NFR framework, one might select responsibility assignments that contribute the most to security goals [CNYM00].
- Using the i* framework, one might identify potentially malevolent agents in the system, analyze vulnerabilities through agent dependency links, and look for counter-measures [LYM03].

The work reported in this chapter makes one step further along those efforts to provide effective, formal guidance in security-driven determination of suitable agents, responsibility assignments, and countermeasures to "unreliable" responsibility assignments (in a sense to be made precise).

Our technique guarantees that critical security goals remain enforced in spite of assignment of requirements or assumptions to potentially malicious or vulnerable agents. The key idea is to identify a subset of the agents of the system and ensure that *the enforcement of critical security goals is only dependent on whether these agents enforce the security requirements/assumptions they are responsible for.* This subset of agents is called trustworthy agents, by analogy with this human-level concept.

We thereby provide a new, security-specific criterion for completeness of security requirements that extends the general criterion in [Yue87]; "*a set of security requirements and assumptions is* complete *with respect to a set of security goals if the subset of requirements and assumptions assigned to trustworthy agents is sufficient to ensure that all the security goals are met*". We further make a distinction between critical and non-critical security goals. *Critical security goals* may never be violated. *Non-critical security goals* may be violated as far as there are some countermeasures that still enforce the higher-level critical security goal they contribute to. Our criterion focuses on the critical security goals.

It is crucial here to understand that within the context of this work, the word "trustworthy" does not refer to human feelings or other socio-technical concepts. Rather, trustworthy software or device agents are to be implemented in a certain way, typically, they must provide appropriate robustness at the language and system layer, the protocol, and cryptographic layer or even physical security such as tamper-resistance for agents implemented

by smart cards, etc. (see Section 2.3). Malicious, potentially malicious or vulnerable agents should not be considered as trustworthy, of course.

Within this chapter, we consider that trustworthy agents never fail. On the bad side, they are single point of failures. On the good side, the technique still proves useful for identifying security requirements. This assumption can be lifted at the price of considering Byzantine systems. This is discussed in the open issue section (Section 5.9).

When talking about human agent, our concept of trustworthiness is not rich enough because no human agent is trustworthy in an absolute sense. For example, to make a payment through a debit card terminal in a shop, the seller might be assigned the responsibility of encoding the payment amount into the terminal. The seller would not be trustworthy for encoding the correct amount, from the viewpoint of the buyer, while being trustworthy from his own viewpoint, and vice-versa. A thirdparty agent entering the right amount might be trustworthy under both viewpoints. On the other hand, genuine e-purses and genuine terminals are trustworthy from the viewpoint of both the seller and the buyer. We therefore consider *relative trustworthiness* from the viewpoint of the agents owning the assets that the critical security goal requires to protect. We refer to these agents as *asset owners*.

Definition (*Asset Owner*): An agent is the owner of an asset iff the agent will be harmed if the asset is exposed.

There is another incentive for enriching our model of trustworthiness with the notion of viewpoint. Even though some devices are trustworthy from the viewpoint of every asset owner, such as the genuine terminals in the e-purse system, different human agents might require different proofs that a given instance is indeed a trustworthy one, or that it might not hurt their security goal. In the case of the e-purse system, a seller should ensure that his terminal is indeed a genuine one (in order to get paid for the money stored in the terminal) and the system should be conceived so that a fake terminal cannot hurt the security of a buyer. An asset can of course have multiple owners. Such case might require an extended analysis of security requirement because several asset owners might have different trustworthy agents.

In this approach we do not consider socio-technical and organizational issues where the e-purse manufacturer or the bank might be crooked and e-purse might be voluntarily (or "unvoluntarily") flawed with backdoors. Interested readers might refer to the specific litterature related to socio-technical issues discussed in Section 2.3.4.7.

In this approach, we do not consider the actual feelings or beliefs of human being, such as trust feelings. In practice, agents in the system lack system knowledge to fully appreciate whether a responsibility assignment is

trustworthy or not. One generally tries to design a system so that real users will feel confident in using it, though they do not have the necessary information about the insight of the system to estimate whether or not it is properly secured. For example, a user of the e-purse system does not have a clue on how the central bank office makes sure that money transfer occurring behind the scenes are handled "securely". Interested readers might refer to the specific literature related to trust feeling issues discussed in Section 2.3.4.7. We consider an abstract model where asset owners know everything about the system. It is the *most knowledgeable asset owner* model. For instance, in the e-purse system, we suppose that each and every user knows everything about the system. In practice, our most knowledgeable user is incarnated in a set of proof obligations regarding the identity of agents who are responsible for security goals.

Definition (*Most knowledgeable asset owner model*): Theoretical model where asset owners are completely informed and aware of all security requirements to be implemented by the system. The purpose of this theoretical model is that one should provide proof to this theoretical agent that his assets are properly protected by any system implementing these requirements.

Such most knowledgeable asset owner model is somewhat the dual view of the most powerful attacker model taken in security engineering. The most powerful attacker model has been presented in Section 2.3.2 in the context of cryptographic protocol analysis and in Section 4.3 when defining the knowledge capabilities of unauthorized agents at application-level.

We want to formally guarantee the enforcement of critical security goals in spite of malicious or vulnerable agents. To achieve this, countermeasures are introduced whenever a security requirement or assumption is under responsibility of an agent not trustworthy from the viewpoint of every concerned asset owner. Such responsibility assignment is called *untrustworthy assignment*. For requirements completeness, we need to make a worst-case supposition that the agent in an untrustworthy assignment might be malicious. For such assignments we introduce countermeasures adapted from obstacle analysis (see Section 3.2.5.4).

In the previous example of untrustworthy assignment for encoding the payment amount, a countermeasure might consist in denying payment if the encoded amount is not the expected one. This new goal might be enforced by displaying the encoded amount and asking the buyer to confirm that the displayed amount is the expected one. The countermeasure resolves the untrustworthy responsibility assignment to the seller, from the viewpoint of the buyer.

Ensuring that the payment amount is equal to the encoded one is under the responsibility of the terminal. Only Genuine terminals should be able to perform payment operation, as will be discussed in Section 5.6.

5.2 Trustworthiness in Requirements Models

This section introduces concepts that are needed for reasoning about trustworthy responsibility assignments. Our technique is aimed at ensuring that critical security goals are enforced in spite of untrustworthy responsibility assignments. This section makes this objective further precise on the basis of the introduced concepts.

We first introduce some lightweight notation in order to represent concepts from the KAOS meta-model. We then define the necessary formal concepts for representing trustworthiness and asset owner. Finally, we formally define our security-specific completeness criterion on the basis of the introduced concepts.

5.2.1 Notation

Throughout this chapter, we use the following predicates to denote relations of the KAOS model:

- Refines(g_1, g_2) **iff** goal g_1 is in the refinement of goal g_2. We assume that there is no alternative refinement, for the sake of simplicity.
- ResponsibleFor(ag, R) **iff** agent ag is prescribed to be the one (and only one) to restrict its behavior by fixing the values of the variables it controls so as to enforce R.
- $Resp(R)$ is a shorthand for denoting the agent that is responsible for leaf goal R

Our analysis applies to AND-portions of a goal graph where each goal is either refined into one refinement or under the responsibility of one agent. In this chapter, we use the notation $P \models Q$ to denote that assertion P entails assertion Q at all times.

5.2.2 A Goal-oriented Model of Trustworthiness

We represent the concept of assert owner through the *wish* relationship. The wish relationship was introduced in the requirements metamodel in

[DvLF93] to capture the interest of agents in goals. We use it to relate a critical security goal to the corresponding asset owners:

Definition (*Wish*): Wish(ag, G) **iff** agent ag is the owner of an asset that goal G requires to protect.

For example, a Customer in a banking system is the owner of his *AccountBalance* asset; she wishes her balance not to be unduly decreased. In our e-signature case study, the *Signature* is an asset owned by the corresponding *Signer*; she wishes any document she does not agree with not to be certified with her signature. The same critical security goal may be wished by multiple asset owners,for example, in the debit card terminal example in Section 5.1, both the buyer and the seller wish the payment amount to be equal to the value of the bought item.

Definition (*critical security Goal*): A critical security goal is a security goal implied in a wish relationship

Which asset owner instance wishes which goal instance is made precise through an instance declaration, for example, in the debit card terminal example:

Wish
Wisher Signer
Wished Avoid[Unagreed Doc Certified By SignBox]
Instance A signer wishes that a document is never certified with her signature by a signature box if she has not agreed on the document.
Formal Def$(\forall\ doc\colon Document,\ sgn\colon Signer,\ sb\colon SignBox)$
Wish$(sgn,$ Avoid[Unagreed Doc Certified By SignBox](doc, sgn, sb))

In the formal declaration of the wish, we refer to particular instances of concepts involved in a goal. This does not introduce second-order into the language because we only reason about the structure of the goal graph, not about the formal assertions defining the goals. In this sense, we reason at the meta-level. The arguments to the goal definition only serve to make further precise the instances concerned with the goal. This borrows from the notations introduced in [Let01].

Wish links entail proof obligations: we need to prove from the viewpoint of asset owners that the enforcement of goal G depends only on whether trustworthy agents, enforce their assigned security requirements/assumptions.

Our goal-oriented notion of relative trustworthiness is captured through a ternary relationship linking a trustworthy agent ag, a critical security goal G, and a wisher agent w:

Definition (*Trustworthy*): Trustworthy(ag, G, w) **iff** , from agent w's viewpoint, agent ag will behave so that the security goal G is enforced. This implies that w believes that ag enforces some of the security requirements he is responsible for.

In our e-signature system, from the viewpoint of a Signer agent, several agents are trustworthy for ensuring that a document will not be certified with her signature when she does not agree with it. These are: herself, her e-ID, and any genuine signature box. This can be made further precise through an instance declaration specifying who is trustworthy for what and from the viewpoint of whom:

Trustworthiness
Trustee e-ID
Trustworthy For Avoid[Unagreed Doc Certified By SignBox]
Viewpoint Signer
Instance From the viewpoint of a signer, her e-ID is trustworthy for ensuring that a document is not certified with her signature if she has not agreed on the document.
Formal Def($\forall$ doc: Document, sgn: Signer, e: e-ID, sb: SignBox)
CardOf(e, sgn)
$\Rightarrow$ Trustworthy$(e,$ Avoid[Unagreed Doc
Certified By SignBox]$(doc, sgn, sb), sgn)$

5.2.3 Representing Trustworthiness Graphically

In the graphical representation of goal model, Wish and Trustworthiness links can be graphically declared in goal models through the ◇ and ✣ link decorations, respectively (see Figure 5.1). Wisher roles in trustworthiness links are indicated by dotted lines.

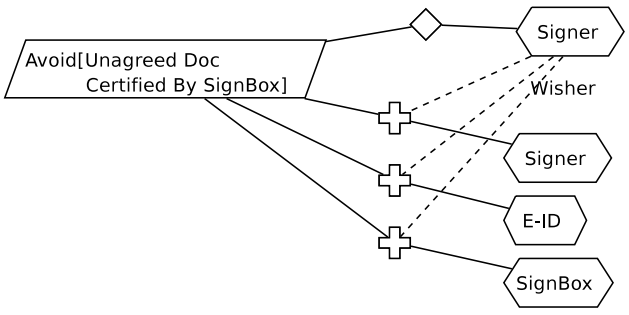

Figure 5.1: Trustworthiness diagram

5.2.4 Completeness of Security Requirements

Our technique is aimed at ensuring that critical security goals are enforced in spite of untrustworthy responsibility assignments. This is a completeness criterion for security-critical application [Yue87]. This section makes this objective further precise.

Definition (*Securely enforced from viewpoint*): A security goal is said to be *securely enforced from the viewpoint of* some agent *ag* if it is enforceable only through requirements or assumptions from the model under the responsibility of agents that are trustworthy from *ag*'s viewpoint. This is formally defined by stating that the conjunction of all the requirements $\{R_t\}_t$ such that Trustworthy(Resp(R_t), G, ag) has the following property: $\wedge_t R_t \models G$.

Definition (*Securely enforced*): A security goal G that is securely enforced from the viewpoint of every wisher agent *ag* defined by Wish(ag, G) is said to be securely enforced.

Definition (*Securely complete*): A requirements model where all *critical* security goals are securely enforced is *securely complete*, or *secure* for short

To summarize, a requirements model is securely complete with respect to a set of critical security goals $\{G_c\}_c$, a set of asset owners and trustworthy agents **iff** for all goals G_c and from the viewpoint of every wisher agent w such that Wish(w, G), the subset of requirements and assumptions R_t assigned to trustworthy agents, that is: such that Trustworthy(Resp(R), G_c, w) enforce all the security goals, that is: $\wedge_t R_t \models G_c$.

This criterion is relative to the critical security goals, the asset owners and the identified trustworhy agents. In particular, a secure model can become insecure if a new wisher agent is added to it.

The definition of *complete* usually used in goal-oriented requirements engineering states that "the requirements together enforce some high-level goal" [vL01], that is: it "only" requires that $\wedge_R R \models G$. Our definition is therefore more restrictive because the classical definition of completeness can rely on all the requirements while our criterion restricts the requirements to the one assigned to trustworthy agents. Some requirements models that are complete might not be securely complete.

5.3 Handling Untrustworthy Assignments

Our aim is to guarantee that a critical security goal is securely enforced in spite of assignments of some of its leaf sub goals to untrustworthy agents. We will refer to such assignments as *untrustworthy assignments*.

Definition (*Untrustworthy Assignment*): An assignment is untrustworthy from the viewpoint of a wisher agent if the responsible agent is not trustworthy from the viewpoint of the wisher agent

We have provided a global completeness criterion over requirements model in the previous section. In this section, we propose a technique for reasoning locally about this global criterion. Our approach is very similar to the goal-oriented approach where the global completeness criterion is implemented locally through local goal refinement completeness as presented in Chapter 3.

The principle of our local approach is that untrustworthy assignments will be resolved by specific countermeasures. The latter are constrained by a set of local meta-level axioms guaranteeing that the critical goal is securely enforced in spite of untrustworthy assignments. This section introduces the axioms on countermeasures. Various classes of countermeasures that meet the axioms are proposed next.

In this section, we first propose a means of representing that a responsibility assignment is untrustworthy. We then introduce the concept of countermeasure to untrustworthy responsibility assignment and a set of local axioms on the requirements model. We then explain why our local axioms together enforce our completeness criterion. Finally, we refine our general notion of countermeasure into intuitive patterns of countermeasure.

5.3.1 Capturing Untrustworthiness of Assignments

The responsibility assignment of a security requirement or assumption might be trustworthy or not, from the viewpoint of some wisher agent, depending on specific conditions. We capture this in the requirements meta-model through the following predicate:

Definition (*Distrust condition*): Distrust(O, LG, v) **iff** condition O can make the assignment of leaf goal LG untrustworthy from the viewpoint of agent v. Agent v wishes some critical security goal which LG contributes to.

Consider the assumption *Avoid[Agreement Notified For Unagreed Doc]*, for example, and the responsibility assignment shown in Figure 5.2. It might be the case that a Signer standing at a signature box with an e-ID inserted

in it is not the one corresponding to the e-ID, making the assignment untrustworthy then. This condition is named *Wrong Signer At SignBox* in Figure 5.2.

Distrust condition Wrong Signer At SignBox
Trustee e-ID
Untrustworthy For Avoid[Agreement Notified For Unagreed Doc]
Viewpoint Signer
Instance It might be the case that a Signer standing at a PC connected to a SignBox is not the one represented by the e-ID inserted into the SignBox
Formal Def($\exists$ sgn, sgn2: Signer, e: e-ID, sb: SignBox, p: PC)
InsertedIn<eid: e, signbox: sb>
$\wedge$CardOf<eid: e, signer: sgn>
$\wedge$At<signer: sgn2, pc: p>
$\wedge$ConnectedTo<pc: p, signbox: sb>
$\wedge$sgn $\neq$ sgn2

A distrust condition is graphically linked to the exposed requirement through a crossed arrow. The viewpoint concerned with the distrust is not represented in the diagram in order to keep figures simple.

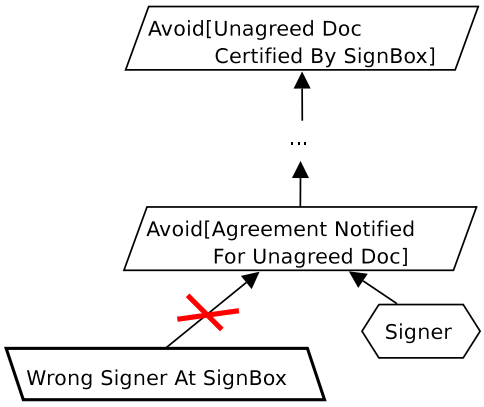

Figure 5.2: Distrust condition

Distrust condition only capture that the responsible agent is not in the set of trustworthy agents from the viewpoint of the asset owner. As explained in Section 5.1, the asset owner agent is taken as a most knowledgeable asset owner.

5.3.2 Countermeasures to Untrustworthy Assignments

A *countermeasure* to a conditionally untrustworthy assignment of some leaf goal is a new goal that ensures the enforcement of some parent security goal, called *reference goal*, which the leaf goal contributes to. We capture this in the requirements meta-model through the following predicate:

Definition (*Countermeasure*): CounterMeasure(CG, LG, D, G_{ref}) **iff** goal CG enforces the higher-level reference goal G_{ref} in spite of the conditionally untrustworthy assignment, under distrust condition D, of the subgoal LG of G_{ref}. Such countermeasure CG is formally defined as follows:

$$CG \models \neg D \vee LG \vee G_{ref}$$

The reference goal must be chosen so that it actually makes sense with respect to our objective, that is, ensuring that the critical security goals are securely enforced. We therefore need to reason over the whole goal graph and over the countermeasures. In order to do this, we introduce the notion of *contribution*. The basic idea is that a goal contributes to another one if it is "below" it in the graph built on the goal refinements and the countermeasures. For clarity, we introduce a direct contribution, capturing this graph and its transitive closure, that we call contribution. As for every concepts in this chapter, contribution might be dependent of a viewpoint. A countermeasure might not contribute to the critical security goal from the viewpoint of every asset owner wishing that goal. In the e-Purse example, checking the amount encoded by the seller contributes, from the buyer's viewpoint, to the goal that the amount is the expected one. From the seller's viewpoint, there is no such contribution.

Definition (*Direct Contribution*): DirectlyContributesTo($G1, G2, w$) **iff** (a) $G1$ refines $G2$ or (b) $G1$ is a countermeasure, from w's viewpoint, to an untrustworthy assignment of leaf goal $G2$ or of another leaf goal through enforcement of reference goal $G2$. This is formally defined as follows: Refines(G_1, G_2) $\vee$ CounterMeasure($G_1, _, _, G_2, v$) $\vee$ CounterMeasure($G_1, G_2, _, _, v$)

The refinement path from a critical security goal to a descendant countermeasure can contain multiple goals; the contribution link from the countermeasure to the critical security goal is then the transitive closure of direct contribution links between adjacent nodes on this path.

Definition (*Contribution*): ContributesTo($G1, G2, w$) **iff** goal $G1$ directly contributes to goal $G2$ or to some goal G' that contributes to goal $G2$. These contribution are relative to the viewpoint of w.

When a countermeasure is added to the goal model, we must make sure

that the reference goal is on every path from the untrustworthily assigned leaf goal to the considered critical security goal. In the terminology of graph theory, the reference goal must *dominate* the distrusted leaf goal over the contribution relation. The following relationship is therefore introduced in the requirements meta-model:

Definition (*Dominate Contribution*): DominatesContribution$(G1, G2, w)$ **iff** from the viewpoint of agent w, $G2$ contributes to $G1$ and if $G2$ contributes to some goal then this goal either contributes to $G1$ or $G1$ contributes to it.

5.3.3 From local security to global security

This section introduces the four local meta-level axioms to be enforced on the requirements model. These axioms further constraint countermeasures to untrustworthy assignments. If these axioms are enforced, the resulting requirements model is securely complete in the sense of Section 5.2.4.

As for the declarations of the *Trustworthy* and *Wish* relationships, these axioms do not introduce second order. They only constraint the structure of the requirements model.

Our *first axiom* on countermeasures defines the conditions under which an untrustworthiness of responsibility assignment obstacle should be introduced in the model. It is the *distrust explicit* axiom:

$$
\begin{aligned}
&(\forall\ ag, w\colon \text{Agent},\ LG\colon \text{LeafGoal},\ G\colon \text{Goal})\\
&\quad \text{Wish}(w, G) \wedge \text{ContributesTo}(LG, G, w)\\
&\quad \wedge \text{ResponsibleFor}(ag, LG) \wedge \neg\text{Trustworthy}(ag, G, w)\\
&\quad \rightarrow (\exists D)\text{Distrust}(D, LG, w)
\end{aligned}
$$

Our *second axiom* defines the conditions under which a countermeasure to an untrustworthy assignment should be introduced so as to ensure a higher-level reference goal, basically, whenever there is an untrustworthiness. It is the *distrust resolved* axiom:

$$
\begin{aligned}
&(\forall\ LG\colon \text{LeafGoal},\ w\colon \text{Agent},\ D\colon \text{DistrustCondition})\\
&\quad \text{Distrust}(D, LG, w)\\
&\quad \rightarrow (\exists CG, G_{ref} : Goal)\text{CounterMeasure}(CG, LG, D, G_{ref})
\end{aligned}
$$

The reference goal must be on all paths from the obstructed leaf goal to the critical security goal it contributes to. Our *third axiom* ensures that the reference goal dominates the contribution of the obstructed goal. It is the *contribution dominated* axiom

$$(\forall\ LG\colon \text{LeafGoal},\ CG, G_{ref}\colon \text{Goal},\ w\colon \text{Agent}, D : \text{DistrustCondition})$$

$$\text{Distrust}(D, LG, w) \wedge \text{CounterMeasure}(CG, D, LG, G_{ref}) \\ \rightarrow \text{DominatesContribution}(G_{ref}, LG, w)$$

Our *fourth axiom* ensures that the countermeasure is not taken too high in the goal graph, so that the critical security goal is enforced, not the goals it contributes to. It is the *no higher than critical* axiom:

$$(\forall\ LG\colon \text{LeafGoal},\ CG, G_{ref}, G_{critical}\colon \text{Goal}, \\ w\colon \text{Agent}, D : \text{DistrustCondition}) \\ \text{Wish}(w, G_{critical}) \wedge \text{ContributesTo}(LG, G_{critical}, w) \\ \wedge \text{Distrust}(LG, D, w) \wedge \text{CounterMeasure}(CG, D, LG, G_{ref}) \\ \rightarrow \text{ContributesTo}(G_{ref}, G_{critical}, w)$$

Theorem 1 *All critical security goals are securely enforced, in the sense defined in Section 5.2.4, if the following conditions hold:*

- *all security goals are Maintain or Avoid goals where the only temporal operator is the outer "always" (□) operator;*
- *the four local axioms presented above are enforced by the requirements model, that is: every conditionally untrustworthy assignment is resolved by a countermeasure that satisfies the meta-level axioms;*
- *every goal in the goal model is either refined or assigned to some agent.*

Proof sketch:

Thanks to the first premise (only one temporal operator: outer "always"), our proofs only manipulate first order concepts, no temporal referencing. We consider an arbitrary position in time. At this time, we consider that only the leaf goals assigned to trustworthy agents are true. We show that the critical security goals are true.

We show first that if the three first axioms are enforced, every goal of the goal graph that contributes to a given critical security goal or that is a critical security goal is either true at this state, or there is a goal that (1) dominates its contribution and (2) contributes to the critical security goal and (3) is true at this state. The proof of this lemma proceeds by induction over the DirectlyContributesTo relation. This relation is well founded as goal graphs are finite directed acyclic graphs.

From this lemma, the proof finishes by showing that the critical security goal is also true because it is covered by the lemma and it does not contribute to a critical security goal.

The proofs can be found in Annex B.1.

5.3.4 Classifying countermeasures

We have proposed a general definition of countermeasure in Section 5.3.2. This definition might be considered as too general to be easily instantiated. We present here three classes of countermeasures to untrustworthy assignments. These classes are specializations of our general definition of countermeasure. We distinguish three classes of countermeasures: strong mitigation, weak mitigation, and untrustworthiness prevention. Other resolution classes might be proposed; they only need to restrict the general definition of countermeasure.

The three proposed classes correspond to well-known obstacle resolution tactics introduced in Chapter 3. Choosing the most adequate countermeasure to resolve an untrustworthy responsibility assignment is up to the requirements engineer. We review these three classes of countermeasure successively.

5.3.4.1 Strong mitigation

In the strong mitigation countermeasure, a higher-level reference goal is enforced in case the untrustworthily assigned leaf goal is not enforced:

$$CG \models LG \vee G_{ref}$$

Typically, this countermeasure class is useful when there is no way to assess neither to ensure that the responsible agent is a trustworthy one. One might assess whether an agent is a trustworthy one via some authentication mechanism and ensuring that only trustworthy agents are able to authenticate themselves. One might ensure that an agent is trustworthy by putting external protection mechanism preventing untrustworthy agent to be in position of being responsible for the requirement. Of course, these mechanisms must be specified by goals and the analysis of secure-completeness must be performed recursively on these goals.

For example, the requirement *Avoid[Doc Without Notified Agreement Sent To SignBox]* is assigned to the PC agent in the goal graph in Figure 5.3. This assignment is untrustworthy from the viewpoint of the signer because no PC is trustworthy from his viewpoint. The countermeasure class used here is a strong mitigation. It is the goal *Avoid[Unagreed Transmitted Doc Signed]*. The reference goal is *Avoid[Unagreed Doc Signed]*. The countermeasure is refined by displaying the transmitted document on the SignBox and asking the user to confirm the agreement on the displayed document.

The countermeasure is fully defined as follows:

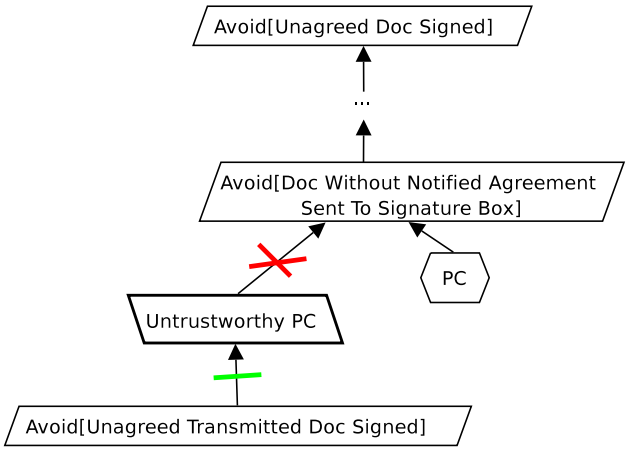

Figure 5.3: Srong mitigation of untrustworthy assignement

Goal Avoid[Unagreed Transmitted Doc Signed]

Definition When a file is transmitted to a sign box, it should not be signed unless the signer represented by the e-ID inserted in the SignBox agrees on the document

Formal Def($\forall$ s: Signer, e: e-ID, f: File, sb: SignBox)
InsertedIn<eid: e, signbox: sb>
$\wedge$CardOf<eid: e, signer: s>
$\wedge$Transmitted<file: f, signbox: sb>
$\wedge \circ$ @Signed<eid: e, signbox: sb>
$\Rightarrow$ Agrees<file: f, signer: s>

5.3.4.2 Weak mitigation

In the weak mitigation countermeasure, a higher-level reference goal is enforced in case the agent assigned to the leaf goal turns out not to be currently trustworthy:

$$CG \models \neg O \vee G_{ref}$$

This countermeasure class requires that it is possible to assess whether the responsible agent is a trustworthy one, typically via authentication mechanisms, similarly to the example discussed for the strong mitigation class presented above. This countermeasure class can be instantiated when the assigned agent may be trustworthy or not depending on some condition on the current state. If the agent will never be trustworthy strong mitigation should be used instead.

For example, the assumption *Avoid[Agreement Notified For Unagreed*

Doc] in Figure 5.4 prescribes that if a Signer does not agree with a document and his e-ID is inserted in a signature box, then the PC connected to the signature box should not be notified that the signer agrees with the document.

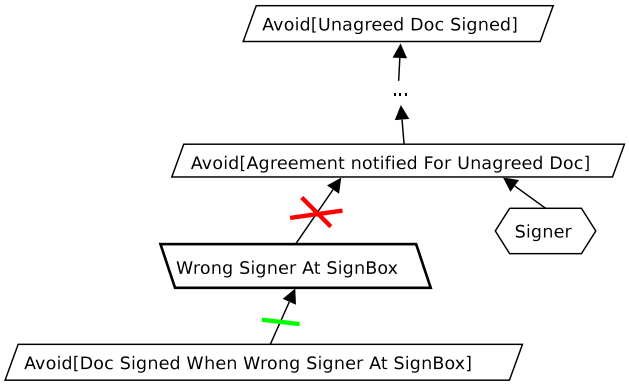

Figure 5.4: Weak mitigation of untrustworthy assignement

This assumption, is assigned to the Signer agent standing in front of the PC. This signer might not be the one represented by the e-ID, as discussed in Section 5.3.1. This is captured by the distrust condition *Wrong Signer At SignBox*. The latter is the only trustworthy signer (as declared in an instance declaration, see Section 5.2.2).

A weak mitigation countermeasure yields the goal *Avoid[Doc Signed When Wrong Signer At Sign Box]* with, as higher-level reference goal, the goal *Avoid[Unagreed Doc Signed]*. The countermeasure goal can then be refined by, for example, a PIN-based authentication subgoals.

The countermeasure goals is fully defined as follows:

Goal Avoid[Doc Signed When Wrong Signer At Sign Box]

Definition A document should not be signed if the signer standing at the PC connected to a SignBox is not the one represented by the ID inserted into the SignBox

Formal Def $(\forall\ s\colon \text{Signer},\ e\colon \text{e-ID},\ \text{sb}\colon \text{SignBox},\ \text{p}\colon \text{PC})$
$\text{InsertedIn}\langle\text{eid}\colon e,\ \text{signbox}\colon \text{sb}\rangle$
$\wedge\text{CardOf}\langle\text{eid}\colon e,\ \text{signer}\colon s\rangle$
$\wedge\text{ConnectedTo}\langle\text{signbox}\colon s,\ \text{pc}\colon p\rangle$
$\wedge \circ\ @\text{Signed}\langle\text{signer}\colon s\rangle$
$\Rightarrow \text{At}\langle\text{signer}\colon s,\ \text{pc}\colon p\rangle$

5.3.4.3 Untrustworthiness Prevention

In the untrustworthiness prevention countermeasure, the responsible agent is enforced not to be an untrustworthy one:

$$CG \models \neg O$$

This countermeasure class requires that it is possible to ensure that the responsible agent is a trustworthy one, typically via physical access control mechanisms preventing untrustworthy agent to be in the situation described by the instance responsibility link.

For example, in the goal graph in Figure 5.5, the requirement *Avoid[Doc Forwarded To Inserted e-Id If No Agreement Notified To SignBox]* is assigned to the SignBox where some e-ID is inserted. Only genuine SignBoxes are trustworthy. Faked ones are not. The countermeasure consists in preventing e-ID from being inserted in non-genuine, faked SignBoxes. This might be enforced by advising signers to insert their e-ID only in SignBoxes delivered by the administration and by preventing the administration from delivering faked SignBoxes.

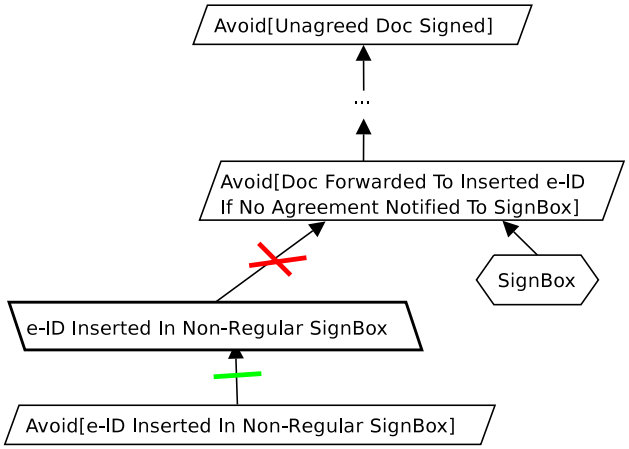

Figure 5.5: Untrustworthiness prevention in assignment

5.4 Ensuring Requirements Completeness

This section presents a process for building requirements models that enforces our requirements completeness criterion. This process ensures that the meta-level axioms are enforced, hence that our completeness criterion is enforced too. The principle is to add countermeasures in a systematic way to prevent or mitigate non-enforcement of security goals in case of untrustworthy responsibility assignments.

5.4.1 Process Overview

Figure 5.6 depicts our process for bringing security in goal models. The steps are ordered by data dependencies and generally intertwined. They are detailed in the next subsections.

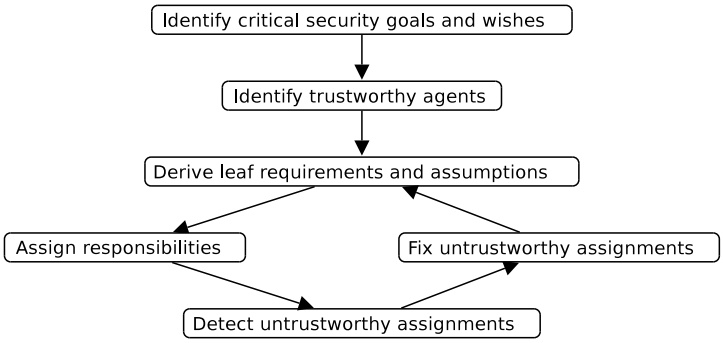

Figure 5.6: Elaborating a secure requirements model

5.4.2 Detailing the Steps of the Process

STEP 1: Identify security goals and wishes

Identifying every critical security goal and every wisher agent is critical; omitting such information can weaken the security of the resulting application as the omitted goals might be violated. The technique described in this chapter only ensures that the identified requirments are complete with respect to the identified critical security goals, wishes and trustworthy agents. This step requires some interactions with domain experts, including experts in security. Nevertheless, heuristics can be used to guide the identification.

Heuristics 1 *Identify critical security goals in the goal graph:*

All goals that are already elicited and integrated in the goal model should be reviewed to determine whether they are (a) referring to asset protection, (b) sufficiently high-level, and (c) precisely definable.

Condition (a) stems from the definition of what a security goal is. Condition (b) is intended to ensure that the refinement paths contain a sufficiently large set of candidate reference goals for countermeasure elaboration. Condition (c) is intended to ensure that the goal specification can be formalized to enable automated reasoning.

Wisher agents are then identified from the critical security goals by (a) identifying the assets those goals prescribe to protect, and (b) looking for the agents that are concerned by these assets.

Heuristics 2 *Identify critical security goals from wisher agents:*

Wisher agents can often be identified from the beginning. Critical assets are central to many applications in banking, insurance, e-commerce, e-government, e-voting, transportation, power systems, etc. Eliciting the agents wishing such assets to be protected might be quite easy. The critical security goals wished by them are the ones prescribing such protection.

For example, any e-signature system involves producers and consumers of agreed documents. The wisher agents are easily identified as the Signer and the Recipient. An obvious critical security goal wished by a *Signer* is that "*a signature box shall not certify as agreed a document not agreed by her*". A critical security goal wished by a *Recipient* is that "*whenever acting upon an agreed document, this agreement is formally certified by a signature box*". The *InstanceWish* declarations are then written accordingly.

This corresponds to the goal *Maintain[Agreement Certified By SignBox When Acting Upon Doc]* and gives rize to the following wish link:

Wish
Wisher Recipient
Wished Maintain[Agreement Certified By SignBox When Acting Upon Doc]
Definition Whenever a recipient acts upon an agreement, this agreement should be certified by a signature box.
Formal Def($\forall$ doc: Document, sgn: Signer, rec: Recipient)
Wish(rec, Maintain[Agreement Certifiable By SignBox
When Acting Upon Doc](doc, rec, sgn))

STEP 2: Identify trustworthy agents

It is crucial not to consider agents to be trustworthy if they are not; this might weaken the security of the resulting application. To reduce the risk of untrustworthy assignments, one trustworthy agent at least should be identified for every wish relationship. There are two ways of achieving this.

(a) Selecting existing agents from the model

Candidates might be the wisher agent herself or an agent that has received some clearance from her. Such clearance can be subject to conditions to be captured by instance declarations of the trustworthy relationship. For example, in e-auction systems, Sellers with a "good" evaluation history are trustworthy from the viewpoint of candidate Buyers. What makes a history "good" depends on criteria decided by the Buyer. Such systems ensure that Buyers can access historical data about Sellers.

(b) Adding agents with built-in trustworthiness to the model

Typical instances of such agents are tamper-proof devices like smart cards.

In the course of identifying trustworthy agents, one should always consider that some agents can be *faked*. Instead of assuming optimistically that all agents of some class are trustworthy, we should state that all *genuine* agents of that class are trustworthy. For example, genuine ATMs and debit card terminals are those being supervised by banks. Recall that our technique does not focus on socio-technical and organisational issues (see Section 2.2). This should be specified through instance declarations of trustworthiness relationships.

In our e-signature case study, the agents that were identified to be trustworthy for the goal *Avoid[Unagreed Doc Certified By SignBox]*, from the viewpoint of the Signer, are depicted in Figure 5.1, namely, the Signer herself, her e-ID, and all genuine SignatureBoxes. An instance trustworthiness declaration for the latter class might be the following:

Trustworthiness
Viewpoint Signer
Trustee SignatureBox
Trustworthy For Avoid[Unagreed Doc Certified By SignBox]
Definition From the viewpoint of a signer, a genuine signature box is trustworthy for ensuring that a document is not certified with her signature by it if the signer has not agreed on the document
Formal Def$(\forall$ doc : Document, sgn: Signer, w: Recipient, sb: SignBox) Genuine(sb) $\rightarrow$ Trustworthy(sb, Avoid[Unagreed Doc Certified By SignBox](doc, sgn, w), sgn)

STEP 3: Derive requirements from goals

This step consists in deriving requirements and assumption through goal refinement process as in the standard goal-oriented methodologies [vL01]. Goals are refined through AND-refinement fine grained goals can be assigned to single agents either software or human.

STEP 4: Assign responsibilities

Responsibilities need be assigned under the constraint that the leaf goals refer to conditions that are monitorable and controllable by the agent [LvL02]. Besides, the leaf goals should be assigned to trustworthy agents whenever possible. When this turns to be not possible, we need to explore countermeasures for enforcing higher-level reference goals.

Trustworthiness tables may be used to guide this step. One such table can be defined for each security requirement; it summarizes all wisher agent classes for that goal, all trustworthy agent classes for that goal, from the corresponding wisher's viewpoint, and all corresponding instance declarations

(see Table 5.1).

Goal Name			
Trustworthy Agents	**Wisher Agents**		
	Wisher Agent class 1	...	Wisher Agent class m
Trustworthy agent class 1	Instance declaration	...	Instance declaration
...	...	...	...
Trustworthy agent class n	Instance declaration	...	Instance declaration

Table 5.1: Trustworthiness Table

For example, consider the assumption *Avoid[Agreement Notified For Unagreed Doc]* in Figure 5.4. The trustworthiness table for this assumption is shown in Table 5.2.

Avoid[Agreement Notified For Unagreed Doc]	
Trustworthy Agents	**Wisher Agents**
	Signer
Signer	Herself
E-ID	Her own
SignBox	All genuine ones

Table 5.2: Trustworthiness table for the assumption *Avoid[Agreement Notified For Unagreed Doc]*

Table 5.2 expresses that the goal *Avoid[Agreement Notified For Unagreed Doc]* contributes to a critical security goal wished by the Signer and that the trustworthy agents for this goal are the signer himself, her e-ID and all genuine signature boxes. Let us apply the *Unique Controller* axiom constraining *Control* links [LvL02] (see Section 3.2.6) and assign the responsibility for this assumption to the Signer standing at the PC. This leads to the responsibility link presented in Section 3.2.5.

STEP 5: Detect untrustworthy assignments

This step is crucial for the goal model to be secure through adequate countermeasures. Checking responsibility assignments against trustworthiness and adding distrust conditions to the model corresponds to enforcing the first meta-level axiom in Section 5.3.2. Tool support for this step is discussed in Section 5.5.

STEP 6: Fix untrustworthy assignments

Untrustworthiness obstacles must be resolved either by modifying the

goal, so as to remove the need for this responsibility assignment, or by fixing the untrustworthy assignment through some appropriate countermeasure, as prescribed by our second axiom.

Modifying the goal model can be performed by exploring alternative refinement or alternative responsibility assignment. The instantiation of the generic countermeasures in Section 5.3.4 must take our third and fourth axioms into account (see Section Section 5.3.2).

For the untrustworthy assignment in the previous section, we might add an authentication-based countermeasure as shown in Figure 5.4. This yields a new security goal to be enforced in turn through security requirements and assumptions obtained by repeating the elaboration process on it. This can lead to nested countermeasures, that is, countermeasures to the obstruction of security leaf goal contributing to countermeasures.

5.5 Automated Detection of Untrustworthy Assignments

Responsibility assignments can automatically be shown to be untrustworthy with the help of verification tools. This section explains how such tools can either show the satisfaction of, or produce a counterexample to, the "distrust explicit" axiom where the outer universal quantification on leaf goals is restricted to some given leaf goal. We report on one experiment conducted with the Alloy analyzer [Jac06]. The experiment presented here was conducted by hand, that is: the problem of checking the trustworthiness of a resposibility assignment has been formulated by hand into the formalism of Alloy. This process can of course be automated [Liu07, Mey07].

5.5.1 Motivation

We want to provide automated support to requirements engineer in the elaboration of requirements model through the process described in Section 5.4.

Testing the "distrust explicit" axiom leaf goal after leaf goal enables us to incrementally test this axiom while elaborating the requirements model and add countermeasures to untrustworthily assigned requirements whenever necessary in order to enforce it. Assessing that the axiom holds for each leaf goal is equivalent to assessing that the axiom holds on the whole requirements model.

If the axiom does not hold on a given responsibility assignment, a distrust condition capturing the ciscumstances in which the agent is not trustworthy must be added to the requirements model. This ensures that the axiom

holds on this responsibility assignment. Here, we only focus on how to automatically assess that a distrust condition must be added to the model. We do not discuss on how to generate the formal definition of this distrust condition automatically. Yet, we present how to illustrate a distrust condition with an automatically generated counter-example. Of course, this distrust condition must be resolved, accordingly to the three other axioms.

Once the "distrust explicit" axiom has been tested on one leaf goal, it does not need to be re-tested again on it. Of course, if a new wisher agent is added to the model or if a trustworthiness relationship is removed, we will be forced to re-assess our "distrust explicit" axiom on every responsibility assignment. Such modifications of the requirements model are not likely to occur, due to the structure of the process presented in Section 5.4.

In this section, we consider the assumption *Avoid[Agreement Notified For Unagreed Doc]* that is under the responsibility of the signer standing at the *SignBox* the agreement is notified. This responsibility assignment is not trustworthy because this signer might not be the one concerned with the agreement. This untrustworthiness should be automatically reported to the user by the goal model editor. The latter should insert the corresponding distrust condition into the goal model, namely: *Wrong Signer At SignBox*. This distrust condition is fully defined in Section 5.3.1.

5.5.2 Process Overview

Automated tools do not support the concept of goals, refinement, agent, etc (at least in the sense they are used here). We therefore need to encode the first axiom and part of the input model into a suitable form that is understood and manageable by the chosen automated tool. This requires a problem translation step. Once the tool solves the problem submitted to it, it generally produces a counter-example, a proof, or nothing (no counterexample and/or no proof found). This output must be brought back into the RE formalism (see Figure 5.7).

In order to assess whether a responsibility assignment is trustworthy or not, we need to make the assumption that the requirements model is complete, that is: nothing is to be added to the requirement. It is possible that a trustworthy responsibility assignment becomes untrustworthy, following from:

- the insertion or update of a wish relationship involving a goal the requirement contributes to;
- the insertion of a refinement such that there is one more critical security goal the requirement contributes to;

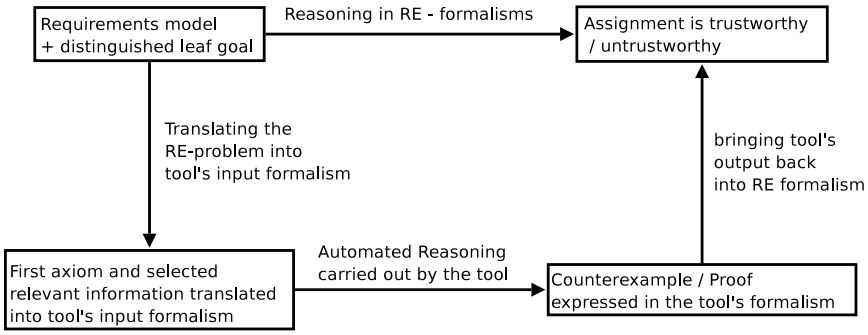

Figure 5.7: Discharging the test of trustworthiness to an automated tool

- the removal of a trustworthy agent for a critical security goal the requirement contributes to.

Similarly, an untrustworthy responsibility assignment might become trustworthy following from:

- the removal or update of a wish relationship involving a goal the requirement contributes to;
- the removal of a refinement such that there the requirement does not contributes to a critical security goal anymore;
- the insertion of a trustworthy agent for a critical security goal the requirement contributes to.

Our approach is to suppose that all the wish, trustworthiness and refinement that might influence on the trustworthiness of the considered responsibility assignment are captured in the model. We say that the model is *closed*. Such closure can typically be computed by applying the Clarck completion procedure (see [RN03] section 10.7). In case an update is done to the model that might change the trustworthiness of the considered assignment, we need to re-assess its trustworthiness. Due to the structure of the process presented in Section 5.4, it is not likely to happen. As far as the process is strictly followed, we can make the assumption that the requirements model is *closed* when testing our restricted axiom. In case this process is not strictly followed, some proofs of trustworthiness must be redone.

5.5.3 Detecting Untrustworthiness with the Alloy Tool

In this section, we show how the Alloy analyzer can be used to show that a responsibility assignment is untrustworthy [Jac00].

Alloy is a structural modeling languages based on first-order logic, for expressing complex structural constraints. The Alloy analyzer is a constraint solver implemented by a SAT-solver that provides fully automatic checking for the Alloy language. It is able to provide an instantiation of a first-order model that violates (or enforces) a given set of assertion. A typical use of Alloy is to test first-order claims for simple counterexamples before investing in a formal proof of it. Alloy has been developed by the Software Design Group at MIT. The Alloy language supports objects, quantifiers, higher arity relations, polymorphism, subtyping, and signatures (Alloy's structuring mechanism). It is considered as one of the most powerful and scalable tool for first-order reasoning. The Alloy analyser has a user interface providing flexible visualization mechanism [Jac06].

Alloy is said to be counter-example-sound because the counterexample it finds are indeed counterexamples to the theorem that invalidate it. It is also said to be counter-example-bounded-complete in the sense that if it does not find any counterexample within the bounds that are given to it, there is indeed no such counterexample within these bounds. On the other side, if Alloy does not find any counterexample to a given claim, it might still be the case that one such counter-example exists outside the search bounds given to it.

Alloy proceeds by propozitionalizing the first order problem by finitely instantiating the input problem based on some bound provided at its input. The propositional problem is then solved by applying efficient SAT-solving techniques.

We have chosen Alloy because it is very simple to interface with, thanks to its rich input language; it is fully automated and quite efficient thanks to recent advances in SAT solving technologies.

The basic principle for having Alloy test the trustworthiness of a responsibility assignment is to feed it with the requirements model and with the first axiom where the outer quantification is restricted to the considered responsibility assignment. The first axiom is the theorem to be refuted by the tool. If Alloy finds a counter-example, this counter-example is a situation where the agent instance responsible for the leaf goal is not a trustworthy one. This is basically an attack on the system-to-be. This attack is to be prevented by the resolution of the distrust condition. If Alloy does not find any counter-example, it might still be the case that one such counter-example exists with less restrictive bounds because Alloy is not counter-example-complete.

The input file for Alloy representing the first axiom test for the responsibility assignment discussed here was produced by hand. The counterexample generated by Alloy is shown in Figure 5.8.

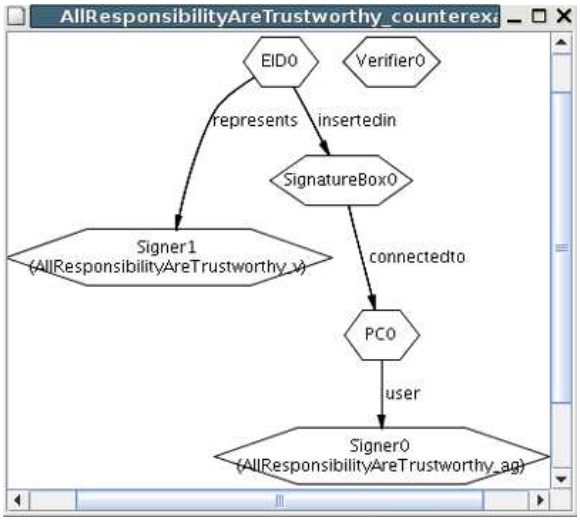

Figure 5.8: Explanation of untrustworthiness given by Alloy

We can see on Figure 5.8 that the Signer standing at `PC0` is not the one represented by the e-ID inserted in the SignatureBox connected to this PC. This is exactly what we expected at first. Notice the bracketed annotation in the hexagons representing `Signer1` and `Signer0`. They capture that the concerned viewpoint is the one of `Signer1` and the responsible agent is `Signer0`, as denoted by the "`_viewpoint`" and "`_responsibleagent`" suffixes appearing after the restricted axiom name. To rephrase this, nothing in our model prescribes that the Signer using a PC is the one represented by the e-ID inserted into the SignBox connected to it.

We now present our encoding of the input problem into the Alloy language. We first present how the object model was encoded, then we focus on the goal model and the theorem encoding.

Alloy supports the basic notion of signature declared with the keyword `sig`. A signature is like a concept class; it can have attributes and be used as members of predicates. Predicates are another basic construct supported by Alloy. They range over signatures. Signatures support inheritance, declared with the keyword `extends`.

The principle for representing the object model is to declare two signature for concept types *Agent* and *Entity* from the KAOS meta model and to declare signature for each agent/entity class similarly, stating which of the meta-model concept they extend. This is declared as follows in the Alloy syntax:

```
//subset of the KAOS meta-model
```

```
sig Agent{}
sig Entity()

//Object model of the signature system:
sig Signer extends Agent{}
sig PC extends Agent{}
sig e-ID extends Agent{}
sig SignatureBox extends Agent{}
sig Recipient extends Agent{}
sig Document extends Entity{}
```

Attribute declaration of a signature should come between the braces at the end of its declaration. For this example, we do not need to represent any attribute of these concepts.

In this particular example, all associations from the object model are binary, they all have one role with a multiplicity of 1:1, and they do not have attributes. We can therefore represent the associations from the object model as attributes of the declared signatures. This lead to the following enriched translation for the object model:

```
//subset of the KAOS meta-model
sig Agent{}
sig Entity()

//Object model of the signature system:
//associations are represented by attributes of the signatures

sig Signer extends Agent{}

sig PC extends Agent{
  at:Signer
}

sig e-ID extends Agent{
  represents:Signer,
  insertedin:SignatureBox
}

sig SignatureBox extends Agent{
  connectedto:PC
}

sig Recipient extends Agent{}
```

```
sig Document extends Entity{}
```

This trick of collapsing the associations and the entities cannot be used when the multiplicities do not allow it or when the associations have attributes. In such more complicated cases, the associations can be encoded as additional signature whose attributes represent the roles of the association.

We now focus on the goal model translation. Goals are to be represented in a similar way than entities and agents. There a still a few differences.

First, we want to reason about instance-level goals, that is: goals in which the outer universal quantification is restricted to specific instances. The reason is that responsibility assignments are specified up to the precise instance of agent. The latter is defined with respect to the identity of some other entities or agents appearing within the goal definition. These can propagate in the goal graph to the identity of the trustworthy agents and wisher agents. Goals have therefore attributes to capture the instance of the agents / entities concerned in the goal.

Second, we need to capture the closure assumption that there is no other critical security goal in the goal graph. In order to capture this closure assumption, we use the Alloy construct called abstract signature. An *abstract signature* means that all instance of this signature are also instance of another signature that extends it. Goals are therefore represented by signature extending some abstract goal signature.

The translation of the goal model into Alloy's input according to these principles is shown below. We only have represented two goals here; these are the considered leaf goal and the critical security goal it contributes to.

```
abstract sig Goal{}
//goal is declared abstract,
//so that every instance of goal must be
//an instance of a specialization of this concept

//goals are to be instantiated because they refer to
//some instance of entities. These entities are mentioned
//as attributes of the goals.

//aanud is a shorthand for
//Avoid[Agreement Notified For Unagreed Doc]
sig aanud extends Goal{
  d1:Document,
  s1:Signer,
```

```
  p1:PC
}

//audcosb is a shorthand for
//Avoid[Unagreed Doc Certified By SignBox]
sig audcosb extends Goal{
  d2:Document,
  s2:Signer,
  v2:Verifier
}
```

We have introduced attributes to goal classes. Because of this we might get the unexpected trouble that there might be two different goal instances with the same attributes. This is obviously undesirable: two instance-level goal of the same class with the same instances are the same. To eliminate this problem, we add the proper facts to the Alloy model. Facts designate assertions that should hold on all provided output of Alloy. The additional facts are defined below:

```
//instance of goals are uniquely defined by
//the goals they instantiate and by their instantiation

fact aanudattrdetermine{
  all g1,g2:aanud{
    g1.d1=g2.d1 and g1.s1=g2.s1 and g1.p1=g2.p1 <=> g1=g2
  }
}

fact audcosbattrdetermine{
  all g1,g2:audcosb{
  g1.d2=g2.d2 and g1.s2=g2.s2 and g1.v2=g2.v2 <=> g1=g2
  }
}
```

We now focus on the relation between goals, and between goals and agents, namely: contribution, responsibilities, wishes and trustworthiness. They can all be represented in the Alloy language as predicates over the signatures previously defined. Alloy predicates are defined with their members and with the condition on them for the predicate to be true or not.

Contributions are derived from refinements and countermeasures; by definition, it is the transitive closure over refinement and countermeasures. As this derivation is trivial, and can be performed by a dedicated routine, we have chosen to perform it out of the alloy tool. The objective is to make

the overall process faster because computing the transitive closure in a tree is linear with respect to the number of nodes in the graph while the time complexity of SAT-solving grows exponentially with the size of the input model. This optimization basically enables us to filter out intermediate and irrelevant goals from the input model. Refinements and countermeasures are therefore not captured in the input file because contribution is directly built from them.

The contribution relationship is defined by the contributes to predicate as follows:

```
pred ContributesTo(g1,g2:Goal){
  (g1 in aanud) and (g2 in audcosb)
  and (g1.s1=g2.s2) and (g1.d1=g2.d2)
}
```

This basically states that within our limited model, the only contribution is between the instance-restricted goals *Avoid[Agreement Notified For Unagreed Doc]* and *Avoid[Unagreed Doc Certified By SignBox]* when they range over the same instance of document and signer.

Similarly, we define the *responsible for* predicate as holding only between a signer agent and a leaf goal *Avoid[Agreement Notified For Unagreed Doc]* and when the signer is the one at the PC connected to the signer box the agreement might be notified to. The formal definition is given in Section 3.2.6. It is recalled below:

Responsibility
Under Responsibilty Of Signer
Responsible Of Avoid[Agreement Notified For Unagreed Doc]
Definition The signer at the PC is the one responsible for not notifying the PC of untrue document agreement
Formal Def($\forall$ sgn, sgn2: Signer, p: PC, sb: SgnBox, e : e-ID)
$\text{At}(sgn, p)$
$\wedge \text{Inserted}(e, sb)$
$\wedge Connected(sb, p)$
$\Rightarrow$ ResponsibleFor(sgn, Avoid[Agreement Notified For Unagreed Doc]($doc, sgn2, p$))

This responsibility assignment is translated into the following Alloy construct:

```
pred ResponsibleFor(ag:Agent,g:Goal){
  g in aanud and ag in Signer
     and some e:EID,p:PC,s:SignatureBox{
        p=s.connectedto
```

```
        and e.insertedin=s
        and p.at:ag
  }
}
```

The wish and trustworthy relationship are defined similarly below.

```
pred Wishes(ag:Agent,g:Goal){
  g in audcosb and ag in Signer and g.s2=ag
}

pred Trustworthy(ag1:Agent,g:Goal,v:Agent){
  g in audcosb and g.s2=v and
  ((ag1 in Signer and ag1=g.s2)
  or (ag1 in EID and ag1.represents=v)
  or (ag1 in SignatureBox))
}
```

Finally, we add an assertion in the Alloy problem to represent our first axiom. In the Alloy language, assertions define things that might be checked. To effectively check that an assertion holds, one must use the `check` construct. This constructs inputs the name of an assertion to be checked and the bounding to be performed by Alloy for propozitionalizing the input problem.

The assertion representing the first axiom is denoted *AllResponsibility-AreTrustworthy*. this assertion is the first axiom without any bounding. These are not necessary for this input file because the only responsibility assignment included in it is the one we want to question. We also have removed the distrust condition from this axiom because the purpose of this checking is to assess the need for a distrust condition when there is no such distrust condition. As a future work, we could also check that the distrust condition indeed captures all circumstances in which the responsible agent is not trustworthy.

```
//this states that all responsibility assignments
//are trustworthy
assert DistrustExplicit{
  all ag,v:Agent,g:Goal,r:Goal{
    (Wishes(v,g) and ContributesTo(r,g)
     and ResponsibleFor(ag,r))
    => Trustworthy(ag,g,v)
  }
}
```

The Alloy instruction for checking this assertion is given below. It imposes bounds of at most 6 instances for each signature.

```
check DistrustExplicit for 6
```

The outcome of this check is shown in Figure 5.8.

Beside this check, we also have performed a test to confirm that the input file itself was not flawed. This is performed by testing that there is at least one instance of the model described by the declared signatures. The principle is to add an assertion that is true and use the `run` command. This command is similar to the check command except that it produces an example such that the assertion holds instead of a counter-example. The Alloy code is as follows:

```
pred example {}
run example for 6
```

In this example, we also tried to add a domain property stating that the signer in front of a PC is the owner of the e-ID inserted in the SignatureBox connected to the PC. With this domain property, Alloy does not find any counterexample. The Alloy code for this additional domain property and the auxiliary Alloy code is shown below:

```
//this is an additionnal domain property
fact EidInsertedByRepresented(){
  all s:Signer, e:EID,p:PC,sb:SignatureBox{
    e.insertedin=sb
    and sb.connectedto=p
    and e.represents=s
    implies p.at=s
  }
}
```

With this additional assertion, Alloy could not find a counterexample to the distrust explicit axiom. The responsibility assignment is indeed trustworthy, thanks to this domain property.

5.5.4 Discussion

In this section, we only focused on the first axiom because it must be repeatedly tested for each responsibility assignment and therefore requires constant attention from the requirements engineer, with a high probability that she will get bored and make mistakes.

A human being might trivially assess the example responsibility assignment considered in this section. Looking at the paragraph introducing it at the beginning of this section, it looks trivial, possibly because it is explained. Nevertheless it is of interest to automatically assess that responsibility assignments are untrustworthy in order to ensure that requirements engineer do not have to perform such trivial reasoning themselves.

The other axioms refers to the countermeasures. As such, they do not require constant attention from the requirements engineer but only time-limited attention when she is about to add a countermeasure to an untrustworthy responsibility assignment. Discharging the testing of the "distrust explicit" axiom is therefore likely to provide the maximal productivity speedup in the elaboration of security requirements.

It might be useful to propose an automated technique for assisting requirements engineer in the elaboration of adequate countermeasure enforcing the other axioms or simply, for assessing whether a newly added countermeasure enforces the other axioms. This is left as an open issue.

Also, another interesting open issue would be to compute the distrust condition via regression techniques similar to the ones used for computing obstacle condition [vLL00]. Here, we only assess that this distrust condition is not false. We could also check that the distrust condition indeed captures all circumstances in which the responsible agent is not trustworthy.

5.6 Evaluation

We have applied our technique to find the security requirements of the e-purse system described in Section 2.1. We have compared our requirements model to the one prescribed by the European Central Bank for such system [ECB03]. This document presents the expectations of the European central bank with regards to the technical security of e-money schemes. It contains a *list of security objectives that should be met by e-money schemes in order to cover risks/threats identified in a comprehensive risk/threat analysis, according to the Common Criteria methodology* [CC99].

5.6.1 Methodology of the Comparison

The document is a protection profile specifying the requirements of e-money systems in an implementation-independent way. It also presents the set of theats from wich the security requirments were identified. This notion of implementation-independence implies that the specified security requirements are not of the same level of abstraction as the ones we are able to find via goal-oriented methods. The requirements appearing in protection-profile

correspond to intermediate-level goals appearing in our requirements models. The requirements appearing in a protection profile require further refinement in order to be assignable to a single agent in the system-to-be. In order to provide a fair comparison, we compared our whole goal graph with the protection profile and only reported requirements that are not mentioned in the protection profile nor derivable from a requirement of the protection profile.

The protection profile also covers the whole system, that is: the payment operation, the loading of e-purses at ATM and the discharging of terminals to seller's bank accounts. Our model only focuses on the payment operation. Of course, our comparison is restricted to this operation too.

5.6.2 Result

The outcome of our comparison is that we could find two security requirements that were not mentioned in this document nor could be derived from the requirements mentioned in this document. How we could identify them is explained in the next section.

The *first* missing requirement prescribes that both the seller and the buyer should be informed about payment completion. A corresponding threat is that a malevolent seller might pretend that payment has not succeeded and not deliver the purchased item to the buyer.

The *second* missing requirement prescribes that a payment operation from an e-purse to a terminal can only take place when the e-purse is inserted into the terminal. A corresponding threat is a terminal-in-the-middle attack. A false terminal can be installed on top of the seller's desk that forward the electrical connection of the inserted card to a genuine terminal hidden under the desk. The false terminal apparently behaves as if it were performing regular payment transactions while the seller can perform any transaction on the hidden genuine terminal.

The requirements in [ECB03] also turned to be less precise than the ones produced with our approach. For example, the following requirement appears in [ECB03]:

```
The integrity of the assets is preserved,
in particular EV amounts.
- Every sub-system preserves the integrity
  of the stored EV amounts.
     [OT.SYS.INT.EVSTORAGE]
- Only authorized transactions can modify
  the EV amount stored in a sub-system.
     [OT.SYS. INT.EVTRANSAC]
```

This requirement does not define what an authorized transaction is nor what integrity means. These terms are left undefined in [ECB03]. The abbreviations `OT.SYS.INT.EVSTORAGE` and `OT.SYS. INT.EVTRANSAC` refer to the taxonomy of security requirements proposed in [CC99].

5.6.3 Analysis

We now explain how our technique was applied to find these missing requirements. We focused on the payment operation in the context of a purchase. We have identified three critical security goals and three wisher agents, which are represented in Figure 5.9. These critical security goals are taken high in the goal graph, but still focusing on the payment operation.

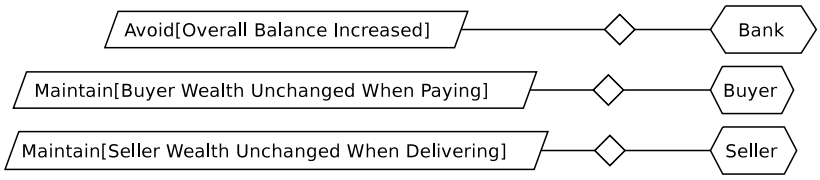

Figure 5.9: Critical Security Goals of Electronic Payment Systems

The goals of Figure 5.9 are defined as follows:

Goal Avoid[Overall Balance Increased]
Definition The sum of the money stored on genuine epurses, genuine terminals and bank accounts should never increase except when physical money is stored on a bank account. In this case, this sum should not increase by more than the amount of money stored on a bank account

Goal Maintain[Buyer Wealth Unchanged When Paying]
Definition The sum of the money stored on the e-purse of a buyer and the value of the good he owns should not be changed when the buyer buys a new item and pays for it with her e-purse

Goal Maintain[Seller Wealth Unchanged When Delivering]
Definition The sum of the money stored on the terminal of a seller and the value of the good he owns should not be changed when the seller delivers an item to a buyer

It is obvious that converse goals must be added to the system for specifying that the wealth of buyers and sellers should not change when they do not perform any transaction. These complementary goals are not considered here for the sake of simplicity. Similarly, the considered model is a simplification over the real world. Here, we make the assumption that buyer

and seller know the value of each item (unlike sandwiches, that might be green inside), that the value of item does not change (unlike sandwiches, that might get green), etc.

Both missing requirements contribute to the critical security goal wished by the buyer. The trustworthy agents from his viewpoint and for this goal are: himself, all genuine e-purses, all regular terminals and the bank infrastructure. This corresponds to the implicit assumptions in [ECB03]. Figure 5.10 presents a fragment of the goal graph rooted at this goal.

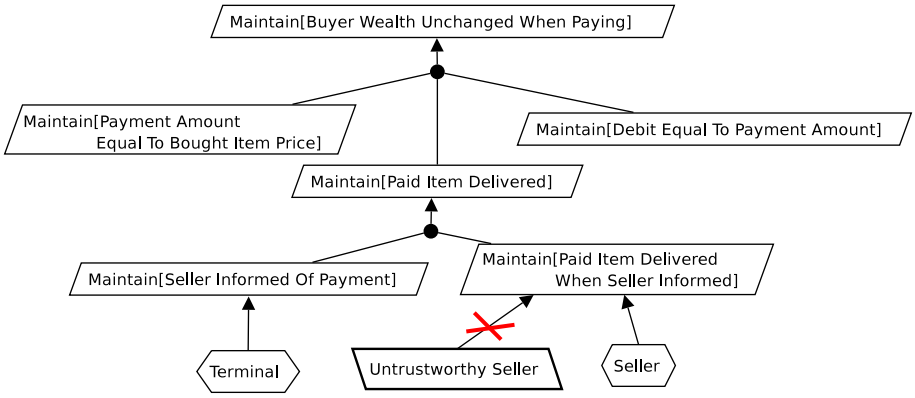

Figure 5.10: Refining the buyer's critical security goal

The goals and requirements appearing in Figure 5.10 are defined as follows:

Goal Maintain[Payment Amount Equal To bought Item Price]
Definition The amount of the payment should be equal to the value of the bought item

Notice that what a payment is only defined as an "electronic event". There is nothing related to the transfer of e-money in the definition of payment up to now. This link between e-money and payment is to be presented in the definition of several goals, among hich, the goal *Maintain[Debit Equal to Payment Amount]* defined below.

Goal Maintain[Paid Item Delivered]
Definition Whenever an item is paid by a buyer, it should be delivered to her

Goal Maintain[Debit Equal to Payment Amount]
Definition Whenever there is a payment from an e-purse to a terminal, the balance of the e-purse should be decreased by the amount of the payment.

This goal is obviously too strong, as it should only be required that

the debit should be at most the value of the bought item. We specify equality here for the sake of simplicity. There is a similar goal related to the increase of the balance of the terminal, that is: *Maintain[Terminal Balance Increased By Payment Amount]* This goal appears in the refinement of the goal *Maintain[Seller Wealth Unchanged When Delivering]*. There is a third goal coming from the critical security goal wished by the bank to ensure that the overall balances of e-purses and terminals does not increase in case of payment. This goal, together with the goals *Maintain[Debit Equal to Payment Amount]* and *Maintain[Terminal Balance Increased By Payment Amount]* together define the effect of performing a payment from an e-purse to a terminal.

The two last goals of Figure 5.10 are defined as follows:

Goal Maintain[Seller Informed Of Payment]
Definition A message should be displayed to the seller on his terminal to inform him that a payment has just been done to his terminal

Goal Maintain[Paid Item Delivered When Seller Informed Of Payment]
Definition When the seller is informed that a payment has just been done to his terminal via a message displayed on it, he should deliver the item that was paid to the buyer

The requirement *Maintain[Seller Informed Of Payment]* or equivalent formulation was not mentioned in the ECB document [ECB03]. There is one more security requirement that prescribes payment information. This requirement is found by looking at the assignment of *Maintain[Paid Item Delivered When Seller Informed]*. This assignment is untrustworthy because the seller is not trustworthy from the viewpoint of the buyer. A possible countermeasure consists in informing the buyer and the seller together that the payment has been done. In an available implementation, the terminal shouts a loud BEEP on payment completion and a different BEEP on payment abort [pro]. It is reasonable to assume that the item will be delivered when buyer and seller have common knowledge of payment completion.

We now present how the second missing security requirement has been found. Figure 5.11 presents a fragment of the goal graph rooted at the goal *Maintain[Payment Amount Equal To Bought Item Price]* introduced in Figure 5.10.

The goals appearing in this figure are defined as follows:

Goal Maintain[Encoded Amount Equal To Bought Item Price]
Definition The amount encoded into the terminal should be equal to the value of the bought item

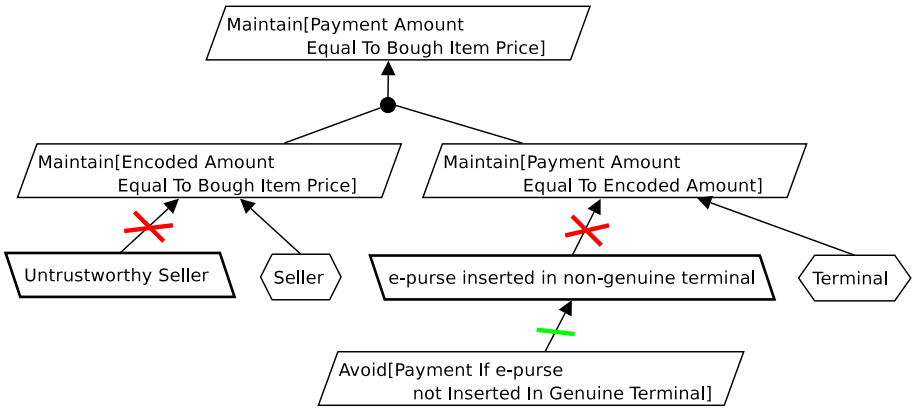

Figure 5.11: Ensuring an adequate payment amount

Requirement Maintain[Payment Amount Equal To Encoded Amount]
Definition The payment amount should be equal to the amount encoded into the terminal

Goal Avoid[Payment If e-Purse not Inserted into Genuine Terminal]
Definition There should be no payment from an e-purse if it is not inserted into a genuine terminal

The two responsibility assignments presented in Figure 5.11 are untrustworthy. The one on the left is similar to the vulnerability discussed in the introduction. It is resolved by adding the following countermeasure:

Goal Avoid[Payment Done When Encoded Amount Not Equal To Bought Item Price]
Definition The payment from a terminal to an e-purse must only take place when the entered amount is equal to the value of the bought item

This goal is refined by requiring that the amount entered by the seller is validated by the buyer. The refining goals prescribe that the entered amount should be displayed on the terminal, that the buyer should only press the "OK" button if the amount is the expected one and that the payment should not proceed while the "OK" button has not been pressed. These goals will be presented in detail in Section 6.1. Yet, we can overview the responsibility assignments coming from them. The responsible agents are the buyer standing at the terminal and the terminal. The buyer is trustworthy in his own viewpoint, though the buyer standing at the terminal might not be the one who owns the e-purse. This is assumed to be the case for this system. There might be concerns about the trustworthiness of the terminal. These are similar to the ones related to the requirement *Maintain[Payment Amount Equal To Encoded Amount]*.

We focus now on the assignment of the requirement of *Maintain[Payment Amount Equal To Encoded Amount]* appearing in Figure 5.11. In the viewpoint of the buyer, trustworthy terminal are the genuine ones, not fake ones. The responsibility assigned to the terminal is therefore untrustworthy because it is assigned to the terminal the e-purse is inserted in, which might be a false one. The countermeasure consists in forbidding payment completion if the e-purse is not inserted into a genuine terminal. This is an example of weak mitigation. This countermeasure is refined as shown in Figure 5.12. None of these requirements were mentioned in the ECB document.

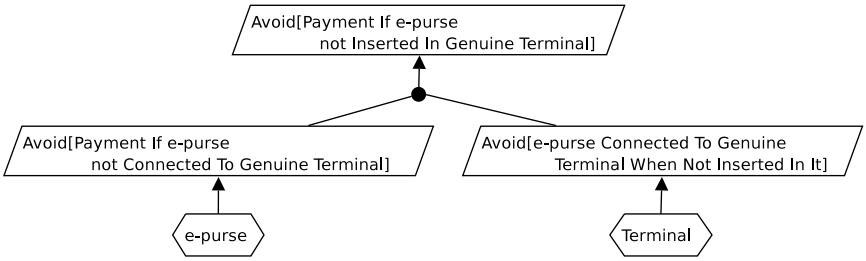

Figure 5.12: Refining the terminal authentication countermeasure

Goal Avoid[EPurse Connected to Genuine Terminal When Not Inserted In It]
Definition An e-purse should not be connected, that is: communicating with a genuine terminal if it is not inserted in it

Goal Avoid[Payment when e-purse not Connected to Genuine Terminal]
Definition There should be no payment from an e-purse if it is not communicating with a genuine terminal

The requirement *Avoid[EPurse Connected to Genuine Terminal When Not Inserted In It]* can be enforced by genuine terminals by measuring the propagation delay of electrical signals or the characteristic impedance of the electrical connection to the smart card it is talking to. If these electrical characteristics are different from the expected ones, a terminal-in-the-middle attack might be ongoing and the genuine terminal should refuse to connect to this e-purse.

5.6.4 Discussion

The requirements discovered via our technique are missing in [ECB03]. They are missing because the corresponding threats were not identified.

Compared to the attack-based approaches, our completeness-based approach is systematic as it targets a completeness criterion. As such, the

requirements engineer is forced to examine all the relevant security issues in the system and provide the necessary security requirements. The word *relevant* and *necessary* are to be taken in the context of our completeness criterion defined in Section 5.2.4.

5.7 Related Work

The explicit representation of trust assumptions within requirements models is advocated in [HLMN06]. These are assumptions about the trustworthiness of system agents. Once identified, such assumptions might appear unacceptable and call for additional security measures or alternative designs. Our approach enables the automatic detection of unacceptable trust assumptions based on trustworthiness assessment for high-level goals.

Trust has also been modeled as an informal non-functional requirement within the i* framework [YL00]. This concept is treated as a softgoal to be satisfied at an acceptable level from the viewpoint of each stakeholder. However, there is no precise way of assessing the coverage and satisfaction of such goals. We propose a formal completeness criterion for this. Moreover, our technique supports a finer-grained notion of alternative decisions based on alternative refinements and assignments.

Bad trust assumptions can be automatically detected in the SecureTropos framework based on a formal model of trust. Agent monitoring is introduced there as a way of overcoming lack of trust among actors [GMMZ05a]. Our three classes of countermeasures for resolving distrust conditions provide additional tactics to remove bad trust assumptions. Some of them produce monitoring and reaction requirements altogether. Our countermeasures may also require trust assumptions to be detected; it is possible to prove that the model does not include any of them. The trust model of [GMMZ05a] is different to our trustworthiness model in that agents responsible for goals might not be able to enforce them. They therefore need to delegate sub goals to other agents, hence there is a notion of trust delegation. We do not have the notion of delegation in our model of trustworthiness because we do not need it, as an agent responsible for a goal is the one that enforces it.

The security engineering approach in the i* framework puts its focus on attackers [LYM03]. The system agents are reviewed to determine what each of them might wish to gain. The approach is informal and excludes threats from outsider agents.

Attacker models commonly used in cryptography share similarities with the view taken in the paper in that all untrustworthy nodes in a network are considered to be potentially malicious [DY81]. The designed protocol

must therefore be made robust against message manipulation that can be performed by those nodes.

5.8 Conclusion

The focus of this chapter was the assignment of responsibilities for security-related goals. Responsible agents might be malevolent or too vulnerable. One should therefore not rely on them for enforcing security goals.

The chapter proposed a goal-oriented model of trustworthiness for reasoning about responsibility assignments. Trustworthy agents are identified from high-level goals, and propagated down to software requirements and environment assumptions. The purpose of the proposed model is to ensure that the enforcement of security goals only depends on whether the trustworthy agents enforce the security requirements or assumptions they are responsible for.

A constructive and compositional process was presented for bringing security into goal models by enforcing this property. Countermeasures are added in a systematic way to prevent or mitigate non-enforcement of security goals in case of untrustworthy responsibility assignments. Some parts of this process were shown to be partially automatable using tools such as Alloy.

Our technique is facilitated by a goal-oriented approach as (a) high-level goals are close to the rationality of potential attackers; trustworthy agents are simpler to identify from high-level goals than from requirements or assumptions, (b) there are much fewer high-level goals than requirements and assumptions to look at for the identification of asset owners and trustworthy agents.

We postulate our technique to scale up easily because it adds very little overhead to the now-classical goal-oriented approaches, which are known to be scalable approaches [vL01, GMMZ04b, CNYM00, GMMZ05a]. We can reasonably assume that the heaviest overhead comes from the enforcement of our first axiom because it requires constant attention from the requirement engineer. This is why we have shown that the enforcement of this axiom can be discharged to automated tools (see Section 5.5).

5.9 Open Issues

Our current goal-oriented trustworthiness model has limitations we are working on. The model does not take agent competence and specialization into account. For example, some agents might be trustworthy for enforcing access control but not trustworthy for enforcing confidentiality goals. Our

current model does not capture the rationale for trustworthiness.

In order to ensure that the goal model adequately captures all desired security concerns, we might end up with a large number of possibly nested countermeasures, that is, countermeasures to non-enforcement of countermeasures. The resulting set of requirements might then be too expensive to implement, and the resulting application might be ineffective because of excessive security measures. Moreover, in case there are only a few trustworthy agents, they might end up with a huge number of responsibilities. Some application might not include any trustworthy agent at all. Last but not least, each trustworthy agent can be considered as a single point of failure. This is because the non-enforcement of a security requirement might cause the critical security goal to be violated. If the security requirement is under the responsibility of an agent that was considered as a trustworthy one, there is no countermeasure to mitigates such non-enforcement.

A possible solution to these four problems would be to weaken our technique to enable the estimation and targeting of some sort of "security level" and "degree of trustworthiness". The main difficulty in doing so is to choose adequate ranking / degree combination rules.

One direction would be to use probabilities [LvL04, LS04]. It is however not quite clear whether probability theory is adequate in our context. For instance, if a system has very little vulnerability, thus a high security level, should we consider that these vulnerabilities would be exploited with a probability of 100%? Such a probability-based approach for quantifying security has been used for monitoring the operational security existing systems [ODK99]. This approach does not apply here because we do not have any operational description of the system, as our technique applies at the requirements elaboration stage.

The question of quantifying security requirements has been analyzed in [HMLN06]. According to this analysis, the major problem with percentage-style quantification of security requirements is the binary nature of the majority of security attacks; in most cases, an attack works or it does not. If an attack does not work the first time, it probably will not work the second time unless the parameters of the attack are changed. On the other hand, if the attack works once (the system is penetrated), then the attack will likely continue working until the vulnerability is removed. The result is that successful attacks can (usually) be repeated as often as the attacker wishes, and even shared amongst attackers. It is difficult to know what use to make of the percentage quantification in these cases.

Another direction is to look at Byzantine systems [LS04]. The completeness criterion would be that the violation of a security critical goal requires the coalition of at least n agents, or agents whose degree of trustworthiness

sum up (or "combine up") to a value higher than some threshold.

We have presented in Section 5.5.4 two open issues of our technique for detecting untrustwworthy responsibility assignment. First, we might explore techniques for automatically computing the distruct condition. The technique proposed in Section 5.5 only assess that this distrust condition is not false. Second, we could also check that the distrust condition indeed captures all circumstances in which the responsible agent is not trustworthy.

Untrustworthy responsibility assignment might be automatically detected and reported by the goal model editor [obj, BDD+98]. For instance, the editor might insert a distrust condition into the requirements model whenever an untrustworthy assignment is added to the model.

Chapter 6

Detecting Divergences in Goal Models

The two previous chapters focus on ensuring that requirements models of security-critical applications are complete. This chapter focuses on the complementary problem: requirements consistency. Requirements assigned to a software agent are inconsistent with each other if they cannot be enforced together by the software agent. In this chapter, we propose a technique for detecting inconsistencies between goals called divergences.

Unlike the two other chapters, the techniques presented in this chapter can also be applied to goals not related to security. Inconsistencies are especially critical in safety-critical systems. This chapter is therefore illustrated on a mine pump system and a train control system from the safety literature in addition to the e-purse system.

This chapter is structured as follows. Section 6.1 illustrates that inconsistencies lead or might lead to vulnerabilities. Section 6.2 introduces our mine pump control system. This example is taken as propositional because our technique for detecting divergences is still limited to the propositional case. Section 6.3 explains why the known formal definition of divergence in [vLDL98] lacks precision. Section 6.4 makes the formal definition fully precise so that an automated technique can be proposed for detecting divergences. Section 6.5 presents our technique for detecting divergences among propositional goals. Section 6.6 presents our prototype implementation and the experiments that have been conducted on it. It served to validate both the new definition of divergence and the two first proposed optimizations by means of benchmarks. Section 6.7 discusses some related work specifically related with divergence detection.

6.1 Inconsistencies as Vulnerabilities

In this section, we show how inconsistencies and vulnerabilities are bound within a requirements model. It is shown on the e-purse system. In this system, we have discovered the two following goals:

Requirement Achieve[Payment Done When e-purse Inserted And Amount Encoded]
Definition A payment must be done from an e-purse to a terminal whenever the e-purse is inserted in the terminal and the amount is encoded in the terminal

Goal Avoid[Payment Done When Encoded Amount Not Equal To Bought Item Price]
Definition The payment from a terminal to an e-purse must only take place when the entered amount is equal to the value of the bought item

The first assertion is a functional requirement under responsibility of the terminal. It has been identified in Section 3.3. The second one is a security goal appearing in a countermeasure to an untrustworthy responsibility assignment. It is identified in Section 5.6. The security goal must be refined because no agent can enforce it. The goal graph is shown in Figure 6.1.

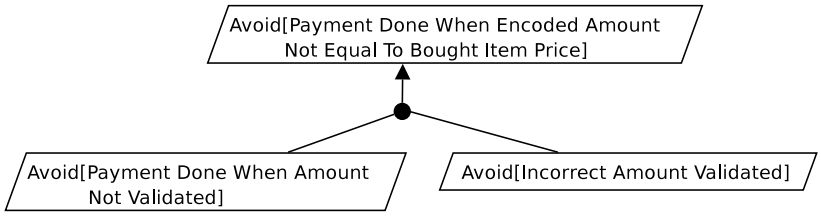

Figure 6.1: Validating the amount encoded in an e-Purse terminal

The subgoals of Figure 6.1 are defined as follows:

Requirement Avoid[Payment Done When Amount Not Validated]
Definition The payment from a terminal to an e-purse must only take place when the "OK" button is pressed

Goal Avoid[Incorrect Amount Validated]
Definition The 'OK' button on a terminal should only be pressed when the entered amount is lower or equal to the value of the bought item

The requirement *Avoid[Payment Done When Amount Not Validated]* is assigned to the terminal, just like the requirement *Achieve[Payment Done When Epurse Inserted And Amount Encoded]* presented at the very beginning of this section. The terminal monitors the 'OK' button, the insertion of

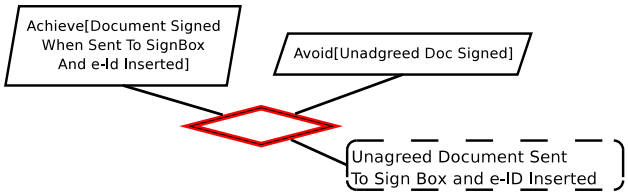

Figure 6.2: Divergence in the e-Signature System

an e-purse into it and the encoding of the amount. It controls the payments.

These two requirements are inconsistent. They cannot be enforced together when an e-purse is inserted into a terminal, the payment amount is encoded into the terminal and it is not validated. This is known as a *boundary condition* [vLDL98]. Whenever this boundary condition presented here occurs, either the security requirement will be violated or the functional requirement will be violated.

If we do not resolve this inconsistency, it might be the case that the terminals that will be produced in later stage of development violate the security requirement. An attack on this terminal is therefore to ensure that the boundary condition is met. In the case of the e-purse system, if the inconsistency is not resolved, the buyer might be able to steal money from the seller by encoding an amount that is larger than the value of the bought item because the terminal might not require the amount to be validated prior to payment completion.

The divergence presented in Section 3.2.5.3 between two goals of the e-signature system is also a vulnerability. This divergence involves the functional goal *Achieve[Document Signed When Sent To SignBox and e-ID Inserted]* and the security goal *Avoid[Unagreed Doc Signed]*. This divergence is recalled in Figure 6.2.

The boundary condition for this divergence is the case where a document is sent to a SignBox and is not agreed by the user represented by the e-ID inserted into the SignBox. In this case, the document should be both signed and not signed. This boundary condition is an attack onto the naive system where documents are automatically signed when sent to a signbox where the e-ID is inserted.

Inconsistencies should therefore be considered as potential vulnerabilities. The inconsistency presented here can be resolved for instance by weakening the functional requirement into *Achieve[Payment Done When Epurse Inserted and Amount Encoded and Amount Validated]*. This weakening must be propagated upwards in the goal graph, to the goals from which this requirement is derived.

The divergences presented here illustrate the problem of inconsistencies between functional and security goals. There might be other inconsistencies, in particular involving security goals only [CNYM00]. For instance, between authentication goals and confidentiality goals, between reactive security goals prescribing that some action must be taken in order to stop an ongoing attack and security goals that require some extra condition to be met prior to taking the reaction. As a last example, there might be conflict between security and safety goals as well. Consider the case of a prison. The prison system should be conceived so that prisoners are evacuated when there is fire at the prison, but the doors of the prison cannot be open to an unauthorized person.

6.2 Introducing a Formally Simpler Example

In this section, we introduce a mine pump example in order to illustrate the concepts introduced in the next sections. The reason is threefold. First, we want to illustrate that detecting inconsistencies is an important problem, not only in the field of security engineering. Second, our technique is still limited to goals declared in propositional temporal logics. The formal declaration of security goals generally involves first order temporal logics. Third, the considered example is sufficiently small to be considered as a whole throughout the presentation.

Our example is a mine pump control system in a coalmine [Jos96, Let01]. A usual problem is that water percolates down into the mine. Pumps are therefore installed at the bottom of the mines to pump water from a collector to the surface. This pump must be turned on when there is a high level of water in the collector. On the other hand, there might be methane in the coalmine and the pump might produce sparks. The latter might cause the methane to ignite. The pump must therefore be turned off when the level of methane is above some critical value. In addition to this, a lamp must be on when there is a high level of water in the collector.

There is an inconsistency in this system because the mine pump must be both turned on and off when there is a critical level of methane and a high level of water at the same time. This inconsistency might be resolved by weakening the requirement related to turning the pump on [vLDL98]. The pump must be turned on when there is a high level of water and the methane level is below critical. Such inconsistency should be detected.

We define here the variables and agents of the MinePump system and we give a fragment of the goal graph on which we can demonstrate our technique. This does not correspond to the real situations, where a goal graph is gradually elaborated, by refining goals, adding new goals, detecting

and resolving inconsistencies. Such process will be presented in Section 6.6.3.

The agents of this mine pump system are:

WaterSensor	that measures the level of water in the coalmine
MethaneSensor	that measures the level of methane in the coalmine
PumpController	that controls whether the pump is switched on or off

The variables of this system are all boolean variables. They are as follows:

CriticalMethane	Captures wether or not the level of methane in the mine pump is above the critical level where sparks can cause it to ignite.
CriticalMethaneSignal	Notifies that CriticalMethane is true.
HighWater	Captures wether or not the level of water is high in the water collector
HighWaterSignal	Notifies that HighWater is true.
PumpOn	Capture whether the pump is on or off.
WaterWarningLampOn	Capture whether the water warning lamp is turned on or off.

The agents control and monitor variables according to the context diagram represented in Figure 6.3. An arrow leading from a variable to an agent denotes that the agent monitors the value of the variable. An arrow leading from an agent to a variable denotes that the agent controls the value of the variable.

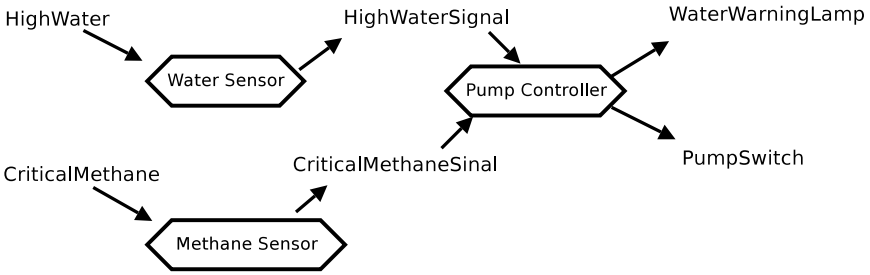

Figure 6.3: Context diagram of a mine pump System

The three following goals can be extracted from the problem statement:

Goal Achieve[Pump Switch On When High Water]
Definition The pump switch must be set to on if there is a high level of water
Formal Def HighWater $\Rightarrow \Diamond_{=2}$PumpSwitchOn

Goal Achieve[Pump Switch Off When Critical Methane]
Definition The pump switch must be set to off if there is a critical level of methane
Formal Def CriticalMethane $\Rightarrow \Diamond_{=2}\neg$PumpSwitchOn

Goal Achieve [Water Warning Light On When High Water]
Definition The Water Warning Lamp must be turned on when there is a high level of water
Formal Def HighWater $\Leftrightarrow \circ$WaterWarningLampOn

The level of methane and water are monitored by the methane sensor and water sensor agents, respectively. They implement the following requirements:

Requirement Maintain[High Water Signal Accurate]
Definition The High Water Signal must be accurate to the high water
Formal Def HighWater $\Leftrightarrow \circ$HighWaterSignal

Requirement Maintain[Critical Methane Signal Accurate]
Definition The Critical Methane signal must be accurate to the critical methane
Formal Def CriticalMethane $\Leftrightarrow \circ$CriticalMethaneSignal

In order to enforce the three high-level goals, the pump controller is made responsible for the three following requirements:

Requirement Achieve[Pump Switch On When High Water Signal]
Definition The pump switch must be set to on if high water level is detected
Formal Def HighWaterSignal $\Rightarrow \circ$PumpSwitchOn

Requirement Achieve[Pump Switch Off When Critical Methane Signal]
Definition The pump switch must be set to off if critical methane level is detected
Formal Def CriticalMethaneSignal $\Rightarrow \circ\neg$PumpSwitchOn

Requirement Achieve [Water Warning Light On When High Water Signal]
Definition The Water Warning Lamp must be turned on when high water is detected
Formal Def HighWaterSignal $\Leftrightarrow \circ$WaterWarningLampOn

The overall goals graph for this system is represented in Figure 6.4.

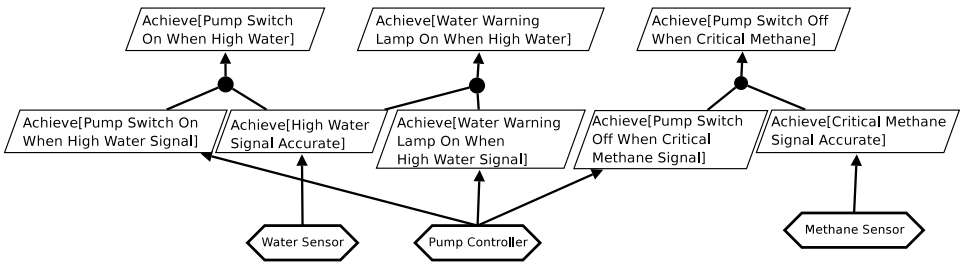

Figure 6.4: Goal graph fragment for the mine pump system

6.3 Problems with the Formal Definition of Divergence

This section shows that the definition of divergence presented in Section 3.2.5.3 is imprecise and must therefore be made fully precise. It is recalled below:

Definition (*divergence*): A divergence between assertions $A_1, \ldots A_n$ occurs within a domain *Dom* **iff** there exists a boundary condition B such that the following conditions hold:

1. $\{Dom, B, \wedge_{i\in[1,\ldots n]} A_i\} \vdash$ false *(logical inconsistency)*
2. for every i: $\{Dom, B, \wedge_{j\neq i} A_j\} \nvdash$ false *(minimality)*
3. there exists a trace S and time position i such that $(S, i) \models B$ *(feasibility)*

A trace is defined as a historical sequence of states. The boundary condition captures a particular combination of circumstances which makes assertions $A_1, \ldots, A_n$ inconsistent if conjoined to it (see conditions (1) and (2)). Condition (3) states that the boundary condition is satisfiable through one behavior at least, that is, there should be at least one trace that establishes the boundary condition.

Consider the goal *Achieve[Pump Switch On When High Water Signal]* and the goal *Achieve [Water Warning Light On When High Water Signal]* These two goals are actually requirements assigned to the *Pump Controller*. They are not inconsistent, as they can be implemented by the *PumpController*. A transition system implementing these two requirements is represented in Figure 6.5. These goals should not be considered as "divergent".

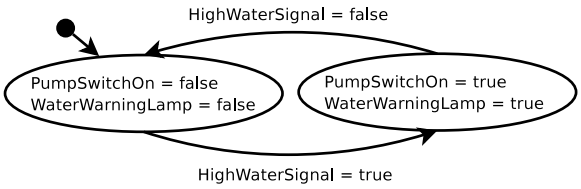

Figure 6.5: Transition system for the *PumpController* agent

However, these two goals are divergent according to the formal definition presented in Section 3.2.5.3. The boundary condition for this divergence is:

$$\Diamond(\text{HighWaterSignal} \wedge \circ\,(\neg\text{WaterWarningLampOn} \vee \neg\text{PumpSwitchOn}))$$

We show below that the three conditions defining divergences hold:

- **Inconsistency:** According to the boundary condition, there must be

a position t such that:

$$\begin{aligned}(H, t) &\models \text{HighWaterSignal} \\ (H, t+1) &\models (\neg\text{WaterWarningLampOn} \vee \neg\text{PumpSwitchOn})\end{aligned}$$

Due to the first statement, the two goals prescribe that:

$$\begin{aligned}(H, t+1) &\models \text{WaterWarningLampOn} \\ (H, t+1) &\models \text{PumpSwitchOn}\end{aligned}$$

The three last statements above are logically inconsistent.

- **Minimality:** removing any one of the two requirements removes the logical inconsistency presented above.
- **Feasibility:** a trace leading to the enforcement of the boundary condition is:

time 0	HighWaterSignal	= true
	WaterWarningLampOn	= false
	PumpSwitchOn	= false
time 1	HighWaterSignal	= true
	WaterWarningLampOn	= false
	PumpSwitchOn	= false

The above boundary condition should not be acceptable because it constrains variables that are actually controlled by the agent we are willing to define, that is the *PumpController*. Here, the boundary condition constraints the *PumpSwitchOn* and *WaterWarningLamp* variables, which are controlled by the *PumpController*. As such, it does not describe *a combination of circumstances that might make the goals inconsistent* as it is expected to (cfr. Section 3.2.5.3).

In general, any set of non-tautological goals $\{G_n\}_{n\in N}$ is divergent with boundary condition $\neg(\wedge_{n\in N} G_n)$ according to the strict definition of Section 3.2.5.3.

6.4 Refining the Formal Definition of Divergence

This section makes the definition of divergence further precise in order to avoid the trivial boundary conditions presented in Section 6.3.

The problem presented in the previous section is first analyzed, which gives rise to the refined definition. The definition is then given. Some auxiliary definitions are given afterwards.

6.4.1 Towards a More Precise Definition of Divergence

The purpose of the feasibility condition was to reject trivial boundary condition such as the one constructed in Section 6.3. We have seen that it does not do the job as it stands. The feasibility condition should therefore be refined.

The original condition is defined as the satisfiability of the boundary condition. The hidden assumption behind satisfiability is that all the agents to be implemented and the environment cooperate in the enforcement of the boundary condition. Actually, some agents might not collaborate in the enforcement of the boundary condition because they are responsible for enforcing requirements contributing to the "divergent" goals.

In a worst-case analysis setting, the actual situation is like a two-player game where the environment does its best to enforce the boundary condition despite any possible reaction of the agents involved in the divergent goals. In this context, the environment is made of *all the agents that will not have any responsibility related to the enforcement of the divergent goals*. The boundary condition should be considered as feasible if and only if this environment can enforce it.

This discussion is very similar to the debate that led to the definition of the realizability condition in the eighties. The purpose was to know whether or not a system with given control/monitoring capabilities could enforce a set of requirements defined in temporal logics. A first version of realizability had been formulated, which reduced to satisfiability of the requirement, just as the feasibility condition presented in Section 3.2.5.3 [CE82, MW84]. This approach is valid when synthesizing a closed system that can control each and every variable referenced in the requirements. In the case of an open system that interacts with its environment, it is widely accepted that realizability must be viewed as a two-player game where the players are the target software and its environment [PR89a, PR89b]. In such game, the target software does its best to enforce its requirements, given its control / monitoring capabilities. The environment can freely choose the value of variables that are not controlled by the target software.

6.4.2 Approximating Controlled Variables at Goal Level through Constrained Variables

Our refined feasibility condition needs to refer to controlled and monitored variables of *those agents that will not have any responsibility related to the enforcement of the divergent goals*.

This refined condition must be assessed when the goals are being identi-

fied without necessarily knowing about responsibility assignments or agents. Agents might not have been identified when divergences are being detected. We therefore attach to the goals some information from which we can approximate the set of controlled and monitored variables of the agents who will be assigned some responsibilities of requirements contributing to these goals. The information consists in tagging the variables referenced by the goal as "intended to be constrained by the goal". These are said to be the *constrained* variables of the goal. This annotation has been introduced in [Let01]. It is recalled and made further precise here.

Definition (*constrained variables*): Constrained variables of a goal denote the set of variables that are intended to be constrained by the goal. These variables are going to be controlled by agents that have some responsibility in the enforcement of the goal.

Constrained variables are generally easy to identify from the goal definition. Usually, they appear in the consequent of the goal. For instance, in the goal *Achieve[Pump Switch Off When Critical Methane]*, the variable *PumpSwitchOn* is the constrained one. Constrained variables are captured in the definition of goals as follows:

Goal Achieve[Pump Switch Off When Critical Methane]
Definition The pump switch must be set to off if there is a critical level of methane
Formal Def CriticalMethane $\Rightarrow \Diamond_{=2} \neg$PumpSwitchOn
Constrained variables: PumpSwitchOn

Constrained variables are formally denoted by the set $Constr(G)$. The set of variables referenced in the formal definition of a goal is referenced to as the *vocabulary* of the goal. It is denoted by the set $Voc(G)$. The variables appearing in this vocabulary, but that are not constrained are referenced to as monitored variables of the goal and denoted $Mon(G)$. We have:

$$Mon(G) = Voc(G) \setminus Constr(G)$$

For instance, in the goal *Achieve[Pump Switch Off When Critical Methane]*, the vocabulary is the set {CriticalMethane, PumpSwitchOn} and the monitored variable is the set {CriticalMethane}.

Constrained variables should be distinguished from variables controlled by agents in that only a single agent can control a variable while many goals can constrain the same variable. The value of a variable should meet all these goals. This is the purpose of requirements derivation and responsibility assignment; we derive requirements to be assigned to agents controlling such variables.

Constrained variables are linked to controlled variables by two axioms,

which formally capture the semantics of the definition of constrained variables. Thanks to these axioms, we will be able to show the link between divergences appearing between high-level goals and non-realizability of requirements derived from divergent goals.

The *first* axiom imposes variables to be coherently constrained along goal graphs. If a variable was referenced and not intended to be constrained by a goal, it should not be constrained by a goal refining it. This is formalized as follows:

$$\begin{aligned}&(\forall G_{\text{child}}, G_{\text{ancestor}} : \text{goal})\\ &\quad \text{ancestor}(G_{\text{ancestor}}, G_{\text{child}})\\ &\quad \rightarrow \text{Constr}(G_{\text{child}}) \cap \text{Voc}(G_{\text{ancestor}}) \subseteq \text{Constr}(G_{\text{ancestor}})\end{aligned}$$

where $\text{ancestor}(G_{\text{ancestor}}, G_{\text{child}})$ denotes that G_{child} is in the transitive closure of the refinement of G_{ancestor}.

The *second* axiom imposes variables constrained by a leaf goal to match the variables controlled by the agent responsible for it. This is formalized as follows:

$$\begin{aligned}&(\forall G\text{: goal},\ \text{ag: agent})\\ &\quad \text{ResponsibleFor}(\text{ag}, G)\\ &\quad \rightarrow (\text{Constr}(G) \subseteq \text{Ctrl}(\text{ag})\\ &\qquad \wedge \text{Mon}(G) \subseteq \text{Mon}(ag))\end{aligned}$$

We can see that these axioms are enforced on the Mine Pump system. For instance, the goal *Achieve[Pump Switch Off When Critical Methane]* constrains the variable *PumpOn*. It is refined by the goal *Achieve[Pump Switch Off When Critical Methane Signal]*. This goal also constrains the variable *PumpOn*. This goal is actually a requirement under the responsibility of the Pump controller agent, which controls the variable *PumpOn*.

The first axiom might seem hard to enforce because it imposes some restrictions on goals that have not been identified, namely, the goals that are to come in the refinement of a given goal. When it is not clear whether a variable of the vocabulary of a goal is constrained or monitored by that goal, one can declare it as constrained in order to ensure that this first axiom is enforced. This is because declaring it as monitored variable forbids us to constrain it in the refinement of the goal. On the other hand, declaring it as constrained variable does not forbid this variable to be monitored in the refinement of the goal. However, we will see in the next section that systematically declaring all variables as constrained variables prevents us from detecting any divergence.

The idea of tagging variables as constrained variables within a refinement also appears in the *B method* [Abr96]. In this method, state machines are refined in a similar way as goals are refined in the KAOS method. A state machine is defined by its input variables, its output variables, its internal states and its transition relation.

6.4.3 A Refined Definition for Divergence

Definition (*divergence*): A divergence between goals $G_1, \ldots G_n$ occurs within a domain *Dom* iff there exists a condition B called a *boundary condition* such that the following conditions hold:

1. $\{\text{Dom}, B, \bigwedge_{1 \leq i \leq n} G_i\} \vdash \text{false}$ (*logical inconsistency*)
2. for every j:
   $\{\text{Dom}, B, \bigwedge_{1 \leq i \leq n \wedge i \neq j} G_i\} \nvdash \text{false}$ (*minimality*)
3. $B \wedge Dom$ is realizable by the environment (*feasibility*)

The environment mentioned in condition 3 monitors all the variables constrained by the goals and controls all the other ones. The environment is assumed to be synchronous. In case a variable is constrained by a goal and monitored by another one, it is taken as being monitored by the environment.

Condition 3 is our refined condition. This condition specifies that the boundary condition must be realizable by the environment. The boundary condition must also be compatible with the domain properties naturally enforced by the environment. We suppose that domain properties are realizable by the environment.

The environment considered in the definition is taken as synchronous because the software agent modeled in the KAOS language are asynchronous in the sense of Section 3.2.8.3 (see Property 1).

This refined definition discards trivial boundary conditions such as the one presented in Section 6.3. In this situation, the goals *Achieve[Pump Switch On When High Water Detected]* and *Achieve [Water Warning Light On When High Water Detected]* constraint the variables *PumpSwitchOn* and *WaterWarningLamp*, respectively. The proposed boundary condition is not realizable by the environment that can only control the *HighWaterSignal* variable.

In the sequel, we often consider that a divergence is a set of goals having the property of being divergent, hence we use the notation $g \in D$ to denote that goal g is involved in divergence D.

6.4.4 Validating the Refined Definition of Divergences

In this section, we want to show that the requirements derived from divergent goals are not realizable by the agents responsible for these requirements. This validates our definition and motivates the usefulness of detecting divergences.

We first show that a minimal set of goal is divergent iff the goals are not realizable by a fictive agent representing the aggregation of the agents that will have some responsibilities related to the enforcement of the divergent goals. We then show that the requirements derived from divergent goals are not realizable by the agents responsible for these requirements.

Theorem 2 *A minimal set of goals D is divergent iff the goals are not realizable by a fictive asynchronous agent representing the aggregation of the agents that will have some responsibilities related to the enforcement of the divergent goals. This fictive agent controls all the variables constrained by the goals in D and monitors all the other variables in the vocabulary of these goals.*

Proof:

- direct implication '$\rightarrow$'

  The direct implication is trivial.

- reverse implication '$\leftarrow$'

  Let be the assertion B defined as the negation of the conjunction of goals in D:

  $$B =_{def} \neg \wedge_{G \in D} G$$

  B is an adequate boundary condition for the divergence D, hence the goals in D are divergent. The three conditions of the definition of divergence are detailed below.

  1. *The conjunction of B and the goals in D is logically inconsistent, by definition of B*
  2. *The conjunction of B and any strict subset of goals in D is not logically inconsistent. This is because the subset of goal is not logically inconsistent, because it is realizable by the agent, which implies that it is consistent.*
  3. *As the conjunction of the goals is not realizable by the asynchronous agent considered here, its negation (that is: B) is realizable by the synchronous environment of this agent, which is the environment considered in the definition of divergence (see Property 1 in Section 3.2.8.3).*

Q.E.D.

From this theorem, we show the link between divergence and realizability of requirements by agents, through the following theorem:

Theorem 3 *Some requirements obtained by refining a set of goals are not realizable together by the agents responsible for them if the following hold together:*

- *there is a divergence involving these goals;*
- *there is no variable that is monitored by a goal, not constrained by any of the divergent goals, and that is constrained in the refinement of one of the divergent goals*

Proof:

Suppose that the requirements derived from divergent goals are realizable by their responsible agents. One can propose a transition system for each agent who is responsible for some of these requirements. A transition system for the fictive high-level agent can be produced from these transition systems. The resulting transition system controls the variables constrained by the "divergent" goals and monitors the other variables in the vocabulary of the goals, because of the axioms linking constrained variables at various level in the goal graph presented in Section 6.4.2.

Hence, these goals are not divergent because they are realizable by this high-level agent (Theorem 2).

Q.E.D.

This theorem motivates the usefulness of detecting divergences between high-level goals. If there is a divergence involving high-level goals, the requirements engineer knows that there will probably be an inconsistency between the requirements derived from these goals. Hence she should first correct the divergence prior to investing time in deriving inconsistent requirements.

The second condition appearing in the theorem shows us that some divergence might not give rise to inconsistent requirements. Such divergence will be referenced to as *spurious divergences*. We illustrate this concept of spurious divergence through an example from a train control system. We consider the two goals fully defined as follows:
Goal Maintain[Door Closed While Moving]
Definition doors should always be closed when the train is moving
Constrained variables:$DoorClosed$
Formal Def$Moving \Rightarrow DoorClosed$

Goal Achieve[Doors Open When Fire]
Definition whenever there is fire onboard, the doors must be opened within two time units
Constrained variables: $DoorClosed$
Formal Def $Fire \Rightarrow \Diamond_{\leq 2} \neg DoorClosed$

According to our definition, these goals are divergent because we cannot enforce both of them in a situation where there is fire onboard and the train keeps on moving for two time units.

This divergence can however be considered as spurious because in case of fire, the train must be stopped prior to doors opening. The train should therefore be stopped within two time units in case of fire. Typically, the goal *Achieve[Train Stopped When Fire]* will be added in the refinement of *Achieve[Doors Open When Fire]*. This new goal constrains the variable *Moving*. This variable can be added to the set of constrained variables of goal *Achieve[Doors Open When Fire]*. This ensures that the goals *Maintain[Door Closed While Moving]* and *Achieve[Doors Open When Fire]* are not divergent.

This feature of the definition of divergence enables us to drive the requirements elaboration process; whenever a divergence is detected, it might either be a spurious divergence and suggest a set of sub goals, like in the example here above, or it can be an inconsistency, like in the mine pump control system.

6.4.5 Organizing Divergences as Root and Implied Divergences

If two goals are divergent, it is very likely that many other goals appearing in the refinements of these goals are also divergent because the definition of refinement ensures that the conjunction of the refining goals together imply the refined goal. If a goal G is involved in a divergence D, it might be the case that there will be a divergence in the set

$$D \setminus G \cup \text{children}(G)$$

where children(G) denotes the goals refining G

In this section, we propose a technique for providing the user with a structured report about the divergences occurring in a goal graph instead of flooding her with information about many related divergences. The risk is that the number of discovered divergences might confuse her, with problems such as "which divergence to resolve first".

For instance, in the mine pump system presented in Figure 6.4, there are four divergences. They are graphically represented by diamonds with a link

to the involved goals in Figure 6.6. The boundary condition of each divergence is informally summarized in the rounded box linked to the diamond.

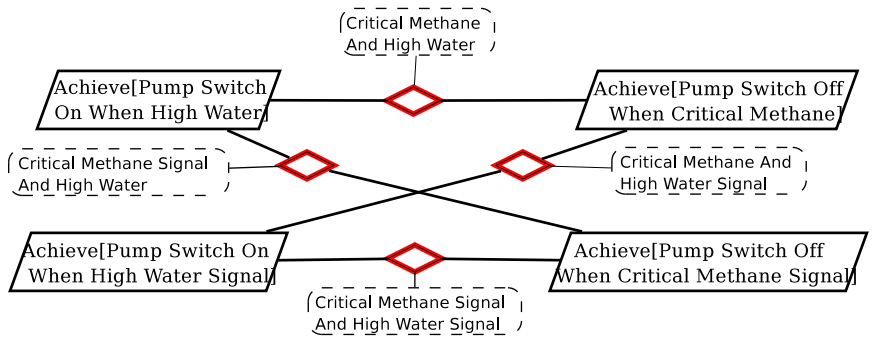

Figure 6.6: Four divergences from the mine pump system

These divergences are obviously related together as they all arise for the same reason: there can be methane and high water at the same time in the coalmine.

We propose a way of grouping divergences into sets of related divergences and represent such sets by a representative divergence. This representative is taken as the one involving the goals occurring as high as possible in the goal graph because high-level goals capture the rational for low-level goals, hence a divergence involving high-level goals might be more related to the rationale behind the divergence. A converse representative would be the goal occurring as low was possible in the goal graph. We first propose the concept of *implied divergence* to capture that a divergence is "lower in the goal graph".

Definition (*Implied divergence*): A divergence D is implied by a divergence D' iff the two following conditions hold:

1. $(\forall G_i \in D)(\exists G_j \in D')\ ancestor(G_j, G_i)$ (*below in goal graph*)
2. $(\forall G_j \in D')(\exists G_i \in D)\ ancestor(G_j, G_i)$ (*the same problem*)

where $G \in D$ denotes that goal G in involved in divergence D.

These two conditions ensure that D can be obtained from D' by replacing some goals of D with goals from their refinement, respectively. Condition (1) ensures that all goals appearing in an implied divergence are below a goal appearing in the implying divergence. Condition (2) ensures that all goals appearing in the implying divergence are above a goal appearing in the implied divergence. This ensures that the implied divergence is not due to an inconsistency introduced in the refinement of one of the goal of the implying divergence.

The "implies" relationships between divergences of Figure 6.6 are graphically represented in Figure 6.7. *Critical Methane And High Water* is a root

divergence. All the other divergences are implied ones.

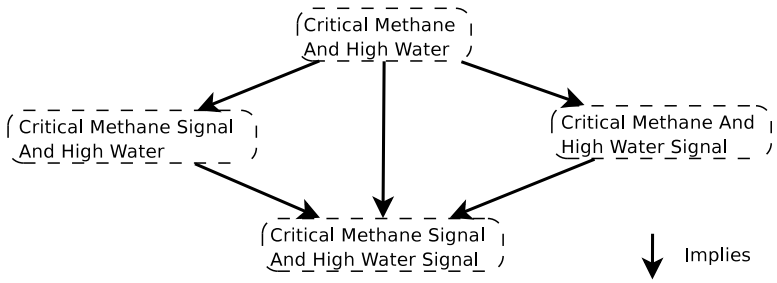

Figure 6.7: Structuring the divergences of the mine pump system

Definition (*Root divergence*): Root divergences are divergences that are not implied by other divergences.

Definition (*Leaf divergence*): Leaf divergences are divergences that do not imply any other divergence.

In order to provide a more succinct output, only root divergences need to be reported initially to the requirements engineer. However leaf divergences can be resolved first by weakening one of the goals involved in them. This weakening is propagated up in the goal graph; this should also remove the other related divergences.

6.4.6 Organizing Boundary Conditions

Divergent goals may accept several different boundary conditions. We need to further characterize them depending on what the boundary condition is used for. This has been mentioned in [vLDL98].

As an example, consider the divergence between the goals *Achieve[Pump On When High Water]* and *Achieve[PumpOff When Critical Methane]*. Three acceptable boundary conditions are:

$$HighWater \wedge CriticalMethane$$

and

$$HighWater \wedge CriticalMethane \wedge \circ CriticalMethane$$

and

$$\Diamond(HighWater \wedge CriticalMethane)$$

Divergences can be resolved by applying standard resolution operators proposed [vLDL98] (cfr. Section 3.2.5.3). One of these operators consists in avoiding the boundary condition. The principle is to add a goal prescribing

that the boundary condition must be avoided. For instance, we may add a goal prescribing that there should not be methane and water together. Installing a methane extractor fan in the mine might enforce this. This tactic does not work when the boundary condition is not the most general one. If we avoid the second boundary condition, we only forbid the presence of methane and water at the initial position of every history.

On the other hand, in order to document a divergence to a user, it is not necessary that the reported boundary condition is the most general one. This is similar to traces delivered by a model checker or counterexamples delivered by automated tools [CGP00, Jac00]. In case a tool discovers that there is some violation of a property it is supposed to check, it only reports one counterexample of this violation, not all the possible violations. The provided counterexample is chosen to be the simplest one; whish is generally translated as the smallest one in term of a metric that can be minimized by the tool.

Definition (*weakest boundary condition*): A boundary condition B of a divergence D involving goals $\{G_i\}_{1\leq i\leq n}$ within domain Dom is the weakest boundary condition if, there is no B' such that:

1. $B \vdash B'$ (*weaker or equivalent*)
2. $B' \nvdash B$ (*not stronger or equivalent*)
3. B' is a boundary condition for divergence D

6.5 Detecting Divergences

This section proposes a technique for detecting divergences in goal models. The technique is made of two steps. The first step extracts sets of goals from the original goal model. The second step tests for the presence of divergences within each of these sets in turn.

This section presents first the problem statement. It then discusses how this problem statement is related to the solution presented in [vLDL98] for testing if a set of goals is divergent. The section then presents the principle of our two-step approach. The steps are then presented successively.

6.5.1 Problem Statement

The problem we want to solve in this section is:

> ***Given*** *a goal model where goals are formally defined in temporal propositional logics and constrained variables are specified for each goal,*
> ***find*** *as many divergences as possible in a fully automated way.*

In the problem statement, "as many divergences as possible" is intentionally left unspecified because our method cannot find all divergences but only a subset of divergence that is formally characterized in Section 6.5.3. Those are the divergences where there is no variable that is constrained by a goal in the divergence and monitored by another goal in the divergence.

The technique presented here is restricted to goals defined in propositional logics and does not support domain properties. Extending the language to include first order constructs and supporting domain properties are left as important open issues for future work. Section 6.9 presents how to extend our technique in order to support first order and domain properties.

We discuss in Section 6.9 another problem statement related to divergence detection where only root or leaf divergences are wanted.

6.5.2 Background on Divergences Detection Techniques

Two techniques were proposed in [vLDL98] for detecting divergences, namely:

- a regression technique for testing if a given set of goals is a divergence. This technique proceeds by chaining clauses extracted from the formal definitions of goals. This technique supports goals defined in first order temporal logics.
- a set of patterns of commonly encountered divergences. Divergences can thus be identified via pattern-matching.

These two techniques were designed for detecting divergences as defined in [vLDL98]. We have refined the definition of divergence, hence these techniques will not detect divergences matching our refined definition. For instance, our refined definition refers to constrained variables. This concept is not handled by these procedures.

On the principle, the regression technique is not guaranteed to terminate, as it searches for an inconsistency in a first order model, which is equivalent to searching for a proof of a theorem formulated in first order logics. Here, we want a fully automated technique; hence we restrict our scope to goals defined in propositional logics. This enables us to build on existing techniques for testing the realizability of temporal assertions.

We also chose not to support domain properties in our technique because it can be easily implemented on top of our technique, as discussed in Section 6.9.

6.5.3 Structuring a Technique for Detecting Divergences

A basic operation for detecting divergences in goal model is to test whether a given set of goals is divergent or not. Testing if a set of goals D is a divergence according to the definition in Section 6.4.3 amounts to testing whether the set of goals in D is realizable by some asynchronous agent, as discussed in Section 6.4.4. Testing the realizability of an assertion by a given agent is a well-known problem with known solutions [PR89a, PR89b, AMPS98, AT04, WDR06, PPS06]. We present a standard solution in Section 6.5.4 and show how a tree-structured trace satisfying the boundary condition can be generated, as an auxiliary result of the realizability test.

In this section, we show how to use this basic block for finding divergences in a goal model. We first propose a naïve approach, show why it is slow and propose another one that builds on a customized realizability-testing procedure.

The first naïve approach for finding all divergences within a set of goals G is to consider all subsets of G and test if these subsets are divergent by applying a realizability-testing procedure.

This naïve approach is time consuming because the realizability-testing problem is solved for each subset of goals and the realizability-testing procedure has a time complexity lying between simple and double exponential time depending on the pattern of the tested temporal assertion [PR89a, PR89b, AT04, WDR06, PPS06].

We take a different approach than the naïve one by proposing a customized realizability-testing technique, on the basis of the standard technique discussed in Section 6.5.4.1. This customized technique computes all the minimal sets of assertions included in a given initial set, such that the assertions belonging to the same subset are not realizable together by the considered agent. This technique requires the sets of controlled and monitored variables to be fully determined prior to the realizability testing, like in any realizability-testing technique.

The difficulty is that the sets of controlled and monitored variables considered in the realizability-testing procedure for testing if a set of goals is divergent depends on the set of divergent goals. Our customized realizability-testing procedure might therefore be useless because the set of controlled and monitored variables corresponding to a set of goals D according to the definition of divergence might not be the one corresponding to any subset of D. We show here that under some conditions, the sets of controlled and monitored variables corresponding to a set of goals D can be used instead of the ones corresponding to any subset of D, with the same result. Under this condition, our customized realizability-testing procedure delivers all the

divergences included in the set of goals submitted to it.

Let be D a set of goals and G a goal that is not included in D. The agent ag is the one that controls all the variables constrained by goals in D and monitors all the other variables of the vocabulary of goals in D. The agent ag' is the one that controls all the variables constrained by goals in $D \cup \{G\}$ and monitors all the other variables appearing in the vocabulary of goals in $D \cup \{G\}$. We want to know the condition such that:

$$\text{Realizable}(\text{ag}', D) \leftrightarrow \text{Realizable}(\text{ag}, D) \tag{6.1}$$

where $\text{Realizable}(\text{ag}, D)$ denotes that the conjunction of the assertions in D are realizable by the agent ag defined by the variable it constraints and the ones it monitors.

This condition is a condition on the sets of controlled and monitored variables of these agents. These sets are defined as follows:

$$\begin{aligned}
\text{Contr}(\text{ag}) &= \cup_{g \in D} \text{Constr}(g) \\
\text{Mon}(\text{ag}) &= \cup_{g \in D} \text{Voc}(g) \setminus \text{Contr}(\text{ag}) \\
\text{Contr}(\text{ag}') &= \text{Constr}(G) \bigcup \cup_{g \in D} \text{Constr}(g) \\
\text{Mon}(\text{ag}') &= (\text{Voc}(G) \bigcup \cup_{g \in D} \text{Voc}(g)) \setminus \text{Contr}(\text{ag}')
\end{aligned}$$

The interface of ag$'$ can be expressed in terms of the one of ag as follows:

$$\begin{aligned}
\text{Contr}(\text{ag}') &= \text{Constr}(G) \cup \text{Contr}(\text{ag}) \\
\text{Mon}(\text{ag}') &= (\text{Voc}(G) \cup \text{Mon}(\text{ag})) \setminus \text{Contr}(\text{ag}')
\end{aligned}$$

From these two assertions, we get that the sets of controlled and monitored variables of ag and ag$'$ can differ in three ways:

1. Some variables can be monitored by ag$'$ and not by ag. These are the variables monitored by G and unknown to the goals in D.
2. Some variables can be monitored by ag and not by ag$'$ because these are controlled by ag$'$ and monitored by ag. This appears when a variable monitored by ag is constrained by G.
3. Some variables can be controlled by ag$'$ and not monitored neither controlled by ag. This appears when a variable was not controlled neither monitored by ag is constrained by G. This case is disjoint with the previous one.

We examine now if these three differences can lead to a violation of (6.1).

- Difference 1 cannot lead to a violation of (6.1) because of Property 4 (cfr. Section 3.2.8.3)

- Difference 2 might lead to a violation of (6.1) because if an assertion is not realizable by an agent, it might become realizable if it is given more control capabilities.
- Difference 3 cannot lead to a violation of (6.1) because of Property 4 (cfr. Section 3.2.8.3)

Condition 2 cannot happen if the set $D \cup \{G\}$ is such that there is no variable that is monitored by a goal and constrained by another one. This is formally defined as follows:

$$(\cup_{g \in (D \cup \{G\})} \text{Mon}(g)) \cap (\cup_{g \in (D \cup \{G\})} \text{Constr}(g)) = \emptyset \tag{6.2}$$

Our approach is therefore to choose maximal sets C of goals such that there is no variable that is constrained by a goal and monitored by another one. This is formally defined in (6.3). If this condition is true, then one can search for divergences in C using the same *Constr* and *Mon* sets.

$$(\cup_{g \in C} \text{Mon}(g)) \cap (\cup_{g \in C} \text{Constr}(g)) = \emptyset \tag{6.3}$$

These sets are called *candidate sets*. They are tested for divergence by applying our customized realizability-testing procedure. Our modified realizability-testing procedure delivers all the divergences included in this set.

The basic algorithm for detecting divergence is presented in Algorithm 2. In this algorithm, *GetAllCandidateSets(AllGoals)* is a function that returns all the maximal subset of *AllGoals* enforcing (6.3). Similarly, *FindAllMinSetOfNonRealizableAssertions* is a function that, given a set of assertions, a set of controlled variables and a set of monitored variables defining an agent interface, returns all the minimal subsets of assertions that are not realizable by the defined agent. The controlled and monitored variables are supposed to cover the vocabulary of every assertion and to be disjoint.

This approach is inherently incomplete in that it cannot detect divergences involving goals that do not enforce (6.3). This is not a real issue because this property is naturally enforced at requirements level. This is because in the KAOS framework, an agent can monitor or control a variable, but not both (see Section 3.2.6) and because at requirements level, variables controlled by agents are the ones constrained by the requirements they are responsible of (see Section 6.4.2).

Section 6.5.4 presents a realizability-testing technique and explains how this technique can be tuned to detect all minimal sets of non-realizable assertions at once within candidate sets of goals enforcing (6.3) and how to generate boundary conditions as an auxiliary result. Section 6.5.5 presents how to choose candidate sets of goals and how to speed up divergence detection by making these sets smaller and/or fewer.

Algorithm 2 {{Goal}}FindAllDivergences(AllGoals:{Goal})

```
% This function returns all the divergences
% included in the set AllGoals
% and that enforce the vocabulary disjointness property (6.3).
% The returned divergences are represented by the sets of divergence
goals.

AllDivergences := ∅
for CandidateSet ∈ GetAllCandidateSets(AllGoals) do
  ControlledVars = ∪_{g∈CandidateSet} Constr(g)
  MonitoredVars = ∪_{g∈CandidateSet} Mon(g)
  AllDivergences ∪= FindAllMinSetOfNonRealizableAssertions(
                                                  CandidateSet,
                                                  MonitoredVars,
                                                  ControlledVars)
end for
return AllDivergences
```

6.5.4 Testing and Documenting the Realizability of Temporal Assertions

This section presents an automated technique for identifying all minimal sets of temporal assertions not realizable together by a given agent. It also presents how to generate a trace illustrating why these assertions are not realizable together. Such traces satisfy the boundary condition (see Section 6.4.6).

Testing a set of goals for divergence amounts to testing the realizability of these goals by a fictive agent built from the set of constrained variables of these goals (see Section 6.4.4). This agent would control the variables constrained by the goals and monitor the variables monitored by the goals.

Testing the realizability of temporal assertions by a given agent is a well-known problem with known solutions [PR89a, PR89b, Sch02a]. Solving this problem in the general case requires double-exponential algorithm, in the worst case [PR90]. Some techniques are able to test the realizability of restricted patterns of temporal assertions with a single exponential time complexity [PPS06, AMPS98, AT04]. Recent techniques perform far better in practice and handle the general case [WDR06].

This section first presents a simplified technique for testing the realizability of temporal assertions, which should be considered as background. It then presents how this technique can be tuned to detect all minimal sets of non-realizable assertions. Finally, it explains how boundary traces can be

generated as an auxiliary result of the realizability test.

We have chosen a simplified technique because it is easier to customize, for instance to generate explanations in case assertions are not realizable, and because the problem of efficiently testing the realizability of temporal assertion is out of the scope of this thesis.

6.5.4.1 Background: Testing the Realizability of Temporal Assertions

In this section, we explain a simplified technique for testing the realizability of a temporal assertion. There are two techniques for testing the realizability of temporal assertions. The first one has been introduced in [PR89a, PR89b, PR90]. The second one improves the first technique and delivers a far better complexity [WDR06]. The technique presented here is a simplification over the first one, with very few adaptations presented explicitly. The second technique is also discussed.

Our simplified technique does not support "eventually" or "until" operators appearing positively in it. The purpose of this restriction is to keep the presentation simple. Our simplified technique can of course be extended so as to handle them. This is briefly discussed in this section.

In this section, we explain a simplified technique for testing the realizability of an assertion ϕ by an asynchronous agent defined by the variables it controls c and the ones it monitors m. These sets are supposed to be disjoint and to cover the vocabulary of ϕ.

The realizability of ϕ is tested by evaluating the definition of realizability:

$$(\exists c_1)(\forall m_1)(\exists c_2)(\forall m_2)\ldots(\exists c_n)(\forall m_n)\ldots H \models \phi \tag{6.4}$$

where H represents a history over m and c such that at any time position t, the value of the controlled and monitored variables is given by m_t and c_t, respectively.

The principle is to evaluate it on a transition system that is equivalent to ϕ. This transition system should not be mixed up with the transition system implementing ϕ. The former is only a temporal tester, which does not distinguish between input and output variables. It is typically taken as the Büchi encoding of ϕ. The latter is a transition system that decides on the value of some variables to enforce ϕ. This one distinguishes between monitored and controlled variables. It only determines the values of the controlled variables and should permanently accept all possible valuations of the monitored variables. Here, we use a simplified procedure where the intermediate transition system is obtained from a clausal form encoding of ϕ

known as the separate normal form [FDP01]. We use this transition system for the sake of simplicity.

Definition (*Separate Normal Form*): The Separate Normal Form (*SNF*) is a clause-like normal form for linear temporal logic. In this thesis, we use a relaxed SNF that has the general form:

$$\wedge_i C_i$$

where each C_i, also referred to as a clause has one of the three following forms:

$$\text{start} \rightarrow P \qquad x \rightarrow \circ P \qquad x \rightarrow \Diamond P$$

where P represent non-temporal assertions, x represents a variable and start is an abbreviation for the assertion $\bullet\neg$true. Variables appearing on the left of next clause are referred to as *memory variables*. The clauses are referred to as *start*, *next*, and *eventuality* clauses, respectively.

Within a history, memory variables of the SNF represent obligations that must be enforced in the next position of the current history (or in some future position in the history for memory variables appearing on the left of eventuality clauses) to ensure that the assertion ϕ is enforced in this current history.

Property 6 *Every LTL formula can be encoded into a SNF that is* equisatisfiable *[FDP01, CRS04]. This might require additional variables to be inserted (the memory variables). In this context, equisatisfiability is formally defined as:*

$$H \models \phi \textbf{ \textit{iff}} \ (\exists s_1, s_2, s_3, \ldots) H^+ \models \phi^{SNF} \tag{6.5}$$

Where s_i represents the value of the n-uple of the memory variables at time i in history H^+; these variables are the one required by the encoding of ϕ into SNF, ϕ^{SNF} represents the SNF encoding of ϕ and history H^+ is identical to history H except that the value of memory variables at any time i of H^+ is given by s_i.

In our simplified technique, we consider the restricted case where there is no eventuality clauses in the SNF. This constraints the input assertion not to have "$\Diamond$" or " $\mathcal{U}$ " operators appearing positively in it because eventually clauses are only generated by the technique described in [CRS04] when there is one such operator.

The SNF is a transition system that can be used as the intermediate transition system for evaluating the realizability of a temporal assertion by

an agent *ag*. The state of this transition system is represented by the n-uple of memory variables from step clauses. The transition system is as follows:

$$\begin{aligned} \text{init} &\subseteq \text{Voc(ag)} \times S \\ T &\subseteq S \times \text{Voc(ag)} \times S \end{aligned}$$

S is the set of all possible assignments to the memory variables of the SNF. The transition T is the conjunction of the next clauses and the initial condition *init* is the conjunction of the start clauses.

We can substitute $H \models \phi$ in the realizability condition (6.4) with its equivalent formula expressed on the SNF encoding (6.5) to get the following equivalent realizability condition:

$$(\exists c_1)(\forall m_1)\ (\exists c_2)(\forall m_2)\ldots(\exists c_n)(\forall m_n)\ldots(\exists s_1, s_2, \ldots)H^+ \models \phi^{SNF} \quad (6.6)$$

Still, (6.6) is difficult to evaluate. Instead of evaluating it, we want to evaluate the following simpler assertion:

$$(\exists c_1)(\forall m_1)(\exists s_1)\ (\exists c_2)(\forall m_2)(\exists s_2)\ldots(\exists c_n)(\forall m_n)(\exists s_n)\ldots H^+ \models \phi^{SNF} \quad (6.7)$$

In order to ensure that (6.7) and (6.6) are equivalent, the transition system represented by the SNF encoding must be deterministic in the sense that at all times, whatever is the past history and the current input and output, there must be at most one next state. There are standard algorithms for adding such property to any transition system and keeping the same models [HU79]. They generally require that the set of states is modified. Here, we suppose that ϕ^{SNF} has been generated so that it is deterministic. Recent techniques enable the efficient direct evaluation of (6.6) and do not require the transition system to be deterministic [WDR06].

We now explain the fix point technique for evaluating (6.7). In terms of transition systems, assertion (6.7) evaluates to false on the described transition system iff it is possible that the transition system is brought into a blocking state, that is: it is possible to force the transition system in a state where we can state that ϕ cannot be enforced whatever is the remaining of the "$\forall\exists$" chain. Such i exists because we only consider ϕ where there is no $\Diamond$ operator.

In order to capture this, we need the following additional definitions:

Definition (*blocking state*): A state s is blocking for a transition system (init, T) iff there is an input i such that the transition system has no output c and new state s' such that $T(s, i, o, s')$.

Definition (*controllable predecessor*): A state s is a controllable predecessor of a set of state S' for a transition system (init, T) iff when the transition system is in state s, the environment can 'force' it to belong to the set S' by choosing the appropriate input to the transition system. Formally, there is an i such that for all o, $T(s, i, o, s')$ implies that $s' \in S'$.

Notice that the set of controllable predecessors of $\emptyset$ for a transition relation T is the set of blocking states of T.

A transition system (init, T) is realizable if there is an initial state that does not belong to the transitive closure of the controllable predecessors of blocking states.

Definition (*divergent states*): The transitive closure of the controllable predecessors of blocking states of a transition relation T is called the divergent states of T. It is formally defined as follows:

$$\begin{aligned} D_T = \mu d.\{s \in X | (\forall c' \in Output(ag)) \\ (\exists m' \in Input(ag)) \\ (\forall s' \in S) T(s, m', c', s') \rightarrow s' \in d\} \end{aligned}$$

Where D_T represents the divergent states of transition relation T and μ is the least fix point operator.

The realizability transition (6.7) is equivalent to the following expressed in term of the divergent states D_T of a transition system [PR89a, PR89b, AMPS98, AT04, Sch02a, WDR06]:

$$\begin{aligned} (\exists c \in Control(ag)) \\ (\forall m \in Mon(ag)) \\ (\exists s \in S) \\ (init(m, c, s) \wedge s \notin D_T) \end{aligned}$$

This gives rise to the technique for testing the realizability of temporal assertion given in Algorithm 3. In this algorithm, *BuildDFA* denotes a procedure for building a deterministic transition system based on an SNF encoding and a determinization [HU79].

This simplified technique is sound and complete for assertions without unbounded eventualities. In this context, "sound" means that if the algorithm reports that an assertion is not realizable by the agent, it is indeed not realizable, "complete" means that if the algorithm reports that an assertion is realizable, it is indeed realizable by the agent. For assertions with unbounded eventualities, it is sound but incomplete because eventualities are captured in so-called eventuality clauses of the SNF transform. These clauses are are discarded.

Algorithm 3 IsAssertionRealizable(A:Assertion,Mon:{Var},Contr:{Var})

% This function returns true iff the assertion A is realizable by
% an asynchronous agent controlling the variables defined in the set *Contr*
% and monitoring the variables in the set *Mon*
% The vocabulary of assertion A is included in the union of the sets *Mon* and *Contr*.
% The sets *Mon* and *Contr* are disjoint.
% Assertion A does not have $\Diamond$ or $\mathcal{U}$ operator appearing positively in it,
% or conversely, $\Box$ or $\mathcal{W}$ appearing negatively.

(Init,T,mem) = BuildDFA(A)
$D_{T,new} := \emptyset$
repeat
  $D_{T,old} := D_{T,new}$
  $D_{T,new} := \{s | (\forall c' \in Contr)(\exists m' \in Mon)(\forall s')$
    $(T(s, m', c', s') \rightarrow s' \in D_{T,old})\}$
  %INV: $D_{T,new}$ *is the set of controllable predecessors of* $D_{T,old}$
until $D_{T,new} = D_{T,old}$
return $(\exists c)(\forall m)(\exists s)(Init(m, c, s) \wedge s \notin D_{T,new})$

6.5.4.2 Identifying all Minimal Sets of Non-Realizable Assertions

In case a set of assertions is not realizable together by a given agent, we want all the minimal subsets of this set such that the assertions belonging to a same subset are not realizable together. Such minimal sets exist as soon as there are assertions that are not realizable together. This is because realizability is monotonic: if a set of assertions is not realizable by a given agent, a superset of assertions is not realizable either by the same agent (see Property 5 in Section 3.2.8.3).

This problem can be formalized as a search in the Boolean lattice built from the set of assertions A given to the realizability-testing algorithm with the subset relation. The elements of this lattice are subsets of A, the top of the lattice is A, and the bottom is $\emptyset$. The elements of the lattice are further partitioned into two sets by the realizability property: the sets of assertions that are not weakly realizable together by a given agent and the sets of assertions that are. The latter includes $\emptyset$.

Efficient algorithms for finding all the minimal elements of a Boolean lattice that enforce a monotonic property P have been proposed in the context of constraint programming [dlBSW03, Jun01]. The basic principle of such algorithm is to "climb up" in the lattice, starting from $\emptyset$, by adding elements of A until the property P is violated. At this point, the last inserted element, possibly together with already inserted ones cause the property to

be violated. The algorithm then proceeds with a form of dichotomy on the sets of elements. These algorithms exploits the fact that incremental checks of the property P can be performed efficiently, that is: once $P(B)$ has been tested where $B \subset A$, it is fast to test that $P(B \cup \{c\})$ where $c \in A$. These are the minimal conflict set algorithms used in Section 4.5.2.7.

Such algorithm applies to the lattice introduced here with the realizability property. The problem of finding all minimal sets of non-weakly realizable assertions can be reduced to searching minimal elements in a lattice in two different ways:

- The first way consists in executing the lattice search on a lattice made of all the assertions with property P set as "non-realizable set of assertions". In this case, the realizability algorithm presented above is called many times by the lattice search algorithm.
- The second way consists in parameterizing every assertion (G becomes $c_g \rightarrow G$ where c_g is the parameter of assertion G), applying the regression technique on all parameterized assertions together and generating a condition on these parameters. This condition captures the combinations of requirements are not realizable together. Put in another form, the valuation set of this condition captures the set of non-parameterized requirements not realizable together. The lattice search algorithm is then applied on this condition to get all minimal set of requirements not realizable together.

The second solution is preferred to the first one because the minimal conflict set algorithms relies on incremental testing of the property. Our regression technique does not make any benefit from this incremental approach, as a complete regression would be performed from scratch every time realizability must be tested. The first alternative would be profitably built on top of a realizability-testing technique supporting incremental realizability testing such as the technique presented in [dAH01].

6.5.4.3 Generating Boundary Traces For Non-Realizable Assertions

When a set of assertions is not realizable together by a given agent, we want a trace that explains why they are so, that is: how to make the assertions inconsistent given the capabilities of the environment surrounding this agent.

If the non-realizable assertions do not include unbounded eventualities, such traces are finite and tree-structured. For instance, the tree represented in Figure 6.8 illustrates a way of making the two following assertions inconsistent: *Achieve[Pump Switch On When High Water]* and *Achieve[Pump*

Switch Off When Critical Methane]. These assertions prescribe that the pump switch must be set to on (resp. off) within two time units if the water level is high (resp. the methane level is critical).

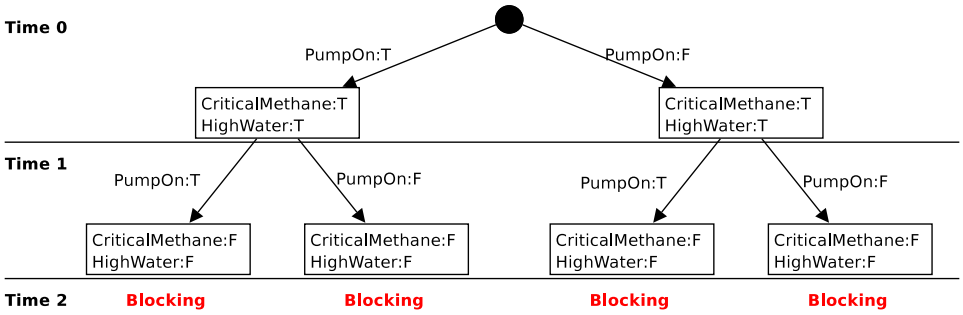

Figure 6.8: A boundary trace for the mine pump system divergence

In Figure 6.8, the edge are labeled with the outputs of the agent and the nodes are boxes containing the input of the agent. Such input bring her into a blocking state. A blocking occurs when there is no edge going out of a box. This tree explains the divergence of the two assertions because every leaf of the tree is a blocking state and the tree explores all possible reactions of the agent to the inputs sent to him. Notice that such representation captures all possible outputs of the agent, while it only captures one input sent by the environment, even if several inputs might bring the agent into a blocking state. In that sense, they are more restrictive than the weakest bounday condition.

Such tree-structured traces can be generated automatically, as an auxiliary result of the realizability testing procedure. This section explains how they can be generated and how they can be made as "small" as possible, so that they are as easy as possible to understand.

Let be $[D_{T,0}, D_{T,1} \dots D_{T,(n-1)}, D_{T,n}]$ the list of sets of divergent states obtained at each iteration of the realizability testing algorithm Algorithm 3. We have that, for any $i \in [0, \dots, n]$, $D_{T,i+1}$ is the set of controllable predecessors of $D_{T,(i)}$, $D_{T,(n-1)} = D_{T,n} = D_T =$ "divergent states of T", and $D_{T,0} =$ "blocking states of T". It is therefore possible to provide the inputs to the agent so that the state of the transition system progresses in this list. When the state of the transition system is included in D_{T1}, there is no possible successor state.

A tree-structured boundary trace can be generated from this list of states by selecting the inputs to send to the agent that forces its state to progress in this list whatever output it provides. The algorithm for generating such boundary trace is presented in Algorithm 4 and Algorithm 5. We assume that the trace is represented by a data structure called

BoundaryTreeNode. BoundaryTreeNode supports one method, that is, the AddBranch($c, m,$ BoundaryTreeNode) method. This method adds a branch to the node; the branch is labelled by the controlled and monitored variables represented by c and m and its root is the third parameter.

Algorithm 4 GetBoundaryTree(T,init,Mon,Contr,$[D_{T,n} \dots D_{T,0}]$)

% This function generates a boundary trace for the assertion
% represented by the transition system (T,init).
% The boundary trace illustrates the non-realizability of this assertion
% by the agent whose interface is defined by the sets *Mon* and *Contr*.
% The D_T, i are such that:
% $D_{T,i+1}$ is the set of controllable predecessors of $D_{T,(i)}$
% $D_{T,0}$ is the set of blocking states of T
% the environment can force that the initial state is in $D_{T,n}$

ToReturn = **new** BoundaryTreeNode
for all $c \in Controlled$ **such that{** $(\forall m \in Mon)(\exists s \in S)$ init(m, c, s) **}**
do
  $m =$ **find** $m \in Monitored$ **such that{**
    $(\forall s \in S)(init(m, c, s) \rightarrow s \in D_{T,n})$ **}**
  $s =$**find** $s \in S$ **such that{**
    $init(m, c, s)$ **}**
  Subtree = Unfold(s,T,Mon,Contr,$[D_{T,(n-1)}, \dots, D_{T,0}]$)
  ToReturn.AddBranch(c,m, Subtree)
end for
return ToReturn

Algorithm 5 Unfold(s,T,Mon,Contr,$[D_{T,j} \ldots D_{T,0}]$)

% This function generates a boundary trace for the assertion
% represented by the transition system (T,init)
% starting at the state s
% The boundary trace illustrates the non-realizability of this assertion
% by the agent whose interface is defined by the sets *Mon* and *Contr*.
% The D_T, i are such that:
% $D_{T,i+1}$ is the set of controllable predecessors of $D_{T,(i)}$
% $D_{T,0}$ is the set of blocking states of T
% the environment can force that the next state state is in $D_{T,j}$
% starting from s

ToReturn = **new** BoundaryTreeNode
for all $c \in Contr$ **such that{** $(\forall m \in Mon)(\exists s' \in S)$ T(s, m, c, s') **}** **do**
  m = **find** $m \in Mon$ **such that{**
    $(\forall s' \in S)(T(s, m, c, s') \rightarrow s' \in D_{T,j})$ **}**
  s' = **find** $s' \in S$ **such that{**
    $T(s, m, c, s')$ **}**
  ToReturn.AddBranch(c,m, Unfold(s',T,Mon,Contr,$[D_{T,j+1}, \ldots D_{T,0}]$))
end for
return ToReturn

Some branches of the generated tree might be unnecessarily deep. For instance, in the boundary trace represented in Figure 6.9, the left-hand side is unnecessarily deeper than the left-hand side. This is because the input sent by the environment in the left-hand side is not chosen to progress as much towards the blocking states as it is in the right hand side.

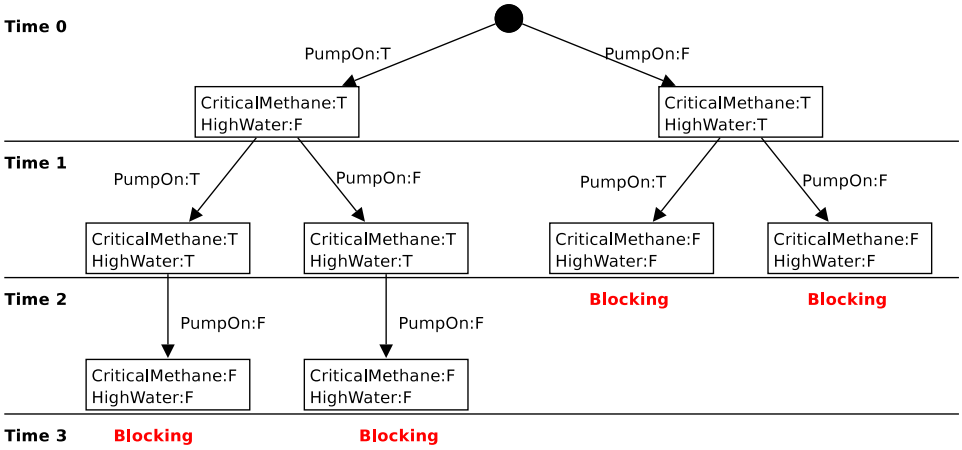

Figure 6.9: An unnecessarily deep boundary trace

In order to limit the depth of every branch, the input sent by the environment (m) should be selected to bring the state of the agent as far as possible in the list of sets of states $[D_{T,n}, \ldots D_{T,0}]$. This can be added in Algorithm 5 and Algorithm 4. A shallow tree generated with such adaptation is shown in Figure 6.8.

The generated tree might be very wide because it covers every possible output of the agent. In case a boundary trace has a branching of two, as in Figure 6.8, the width of the tree is 2^{depth}. In order to provide a more compact representation, similar branches of the tree can be explored and displayed together. This requires that some Boolean values in the output of the agent are replaced by the symbolic value "any". For instance, Figure 6.10 represents the boundary trace of Figure 6.8 where similar branches are displayed together.

A branch starts with the agent's output. Two branches starting at the same position are similar if (1) the disjunction of the first output of the agent at each branch is equivalent to some affectation of the output variables made using the symbolic value "any"; (2) there is an input of the agent that brings the agent to the same internal state for each of these branches; (3) this input progresses towards the blocking. The first condition aims at displaying readable "any" values on the screen. The second aims at ensuring that the two merged branches can share the same continuation. The last one ensures that the algorithm terminates.

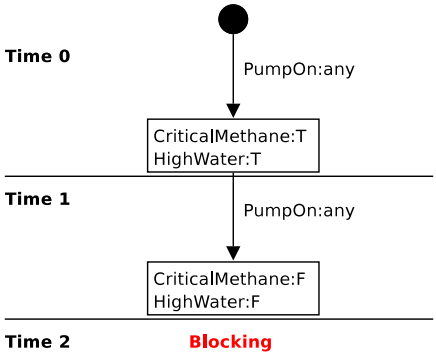

Figure 6.10: A shallow and narrow boundary trace

The two size minimization techniques presented here are local optimization schemes. They do not guarantee that the delivered tree will be the smallest possible.

6.5.5 Choosing Candidate Sets of Goals

This section explains how to chose the candidate sets of goals submitted to the divergence detection procedure presented in the previous section. This section also presents three extensions for making these sets smaller and fewer. The objective of these extensions is to make the overall process of divergence detection faster.

Section 6.5.5.1 presents the basic technique for choosing candidate sets of goals, according to the principles set up in Section 6.5.3. Section 6.5.5.2 presents a technique for splitting candidate sets, based on the vocabulary of goals. Section 6.5.5.3 presents an extension for making divergence detection incremental. This extension is integrated within the one for choosing candidate sets. Section 6.5.5.4 presents a technique for bounding divergence detection. The principle is to look for divergences involving fewer goals, typically two or three, and to reduce candidate sets accordingly.

6.5.5.1 Basic Technique

Candidate sets of goals should:

1. Enforce the vocabulary disjointness property (6.3) that no variable is monitored by a goal and controlled by another one with these two goals being included in the same candidate set. In this section, we denote this assertion by the predicate *MonAndconstrDisjoint* over set of goals.

2. Be such that every divergence enforcing the vocabulary disjointness property (6.3) is included in at least one candidate set of goals, in order to keep the bounded-completeness discussed in Section 6.5.3.

3. Be as small as possible and as few as possible, to spare time in the regression technique.

A first solution, for choosing the candidate sets of goals from the overall set of goals of a goal model is presented in Algorithm 6. The procedure relies on constraint solving techniques represented by the keywords "**findall**" and "**suchthat**" [Sch02b]. The purpose of constraint programming is to efficiently compute some or all the values that match a given set of constraints. A value can be an integer or, as in here, a subset of a given set. We do not represent the technicalities of constraint programming in Algorithm 6. The vocabulary disjointness constraint (6.3) is denoted by the *MonAndConstrDisjoint* predicate. The maximality constraint is denoted by the *Maximal* predicate.

Algorithm 6 {{Goal}}: GetAllCandidateSetsOfGoals(AllGoals:{Goal})

```
return findall CandidateSetOfGoal ⊆ AllGoals
       suchthat {
           MonAndConstrDisjoint(CandidateSetOfGoal)
           Maximal(CandidateSetOfGoal)
       }
```

Candidate sets of goals delivered by this first solution are all the maximal sets of goals enforcing the vocabulary disjointness property. This meets the two first objectives presented in this section.

6.5.5.2 Pruning on Goal Vocabulary

This section presents a technique for pruning candidate sets of goals based on their vocabulary. The objective is to split candidate sets of goals into smaller sets.

Two divergent goals constrain a common variable to opposite signs. For instance, consider the two goals *Achieve[Pump On When High Water]* and *Achieve[Pump On When Pump Button Pressed]*. The latter is defined by:

$$\text{PumpButton} \Rightarrow \circ\text{PumpOn}$$

These two goals are not divergent with each other because the constrained variable *PumpOn* is constrained positively by both goals.

Similarly, the goals *Achieve[Pump On When High Water]* and *Achieve[Water Warning Light On When High Water]* are not divergent with each other because they do not constrain any common variables.

In order to extend this observation to divergences involving more than two goals, we need to introduce the concept of constrained variable opposition graph.

Definition (*constrained variable opposition graph*): A constrained variable opposition graph is a graph where nodes are goals and there is an edge linking two goals iff there is a variable that is constrained positively by one of them and negatively by the other one.

Recall that our technique is limited to boolean variables. A boolean variable is *constrained positively* (resp. *negatively*) by a goal if it is constrained by the goals and if it appears within an odd (resp. even) number of negations. These negations may either be explicit "not" operator or implicit, that is: hidden within the "implies" or the "until" operator. A variable can be constrained both positively and negatively by a goal. The opposition graph can be constructed via syntactical analysis of the formal definition of goal. For instance, the opposition graph of the mine pump system is represented in Figure 6.11.

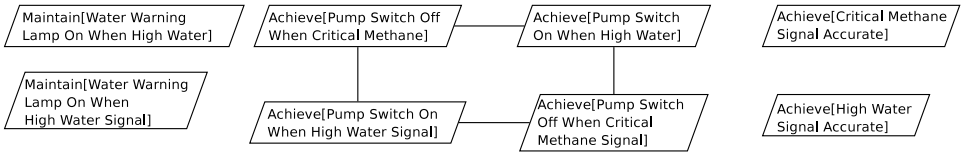

Figure 6.11: Opposition Graph of the Mine Pump system

The property above for two divergences involving two goals is generalized to n-ary divergence by observing that the opposition sub graph inducted by a divergence is connex. This *induced subgraph* is the one made of the goals involved in the divergence and the edges that connect these goals in the opposition graph of the whole requirements model.

Figure 6.12 presents the opposition graphs for three propositional non-temporal requirements models. The goals are represented by their formal definition. All variables are constrained. In the opposition graph of Figure 6.12(i), we can see that goals $\neg a$ and $\neg b$ are not divergent because they are not connected in the opposition graph. From the opposition graph, we can deduce that we do not need to apply the regression technique on the set $\{\neg a, \neg b\}$. We only need to test the realizability of these goals separately, which is quicker. Once the goal $a \vee b$ is added, as in Figure 6.12(ii), we need to test the set $\{\neg a, \neg b, a \vee b\}$ for realizability because these goals belong to the same connex component of the opposition graph. The opposition graph

is an over-approximation of divergences as shown in Figure 6.12(iii). In this figure, another goal is added. This goal is not divergent with the other ones, yet it belongs to the same connex component.

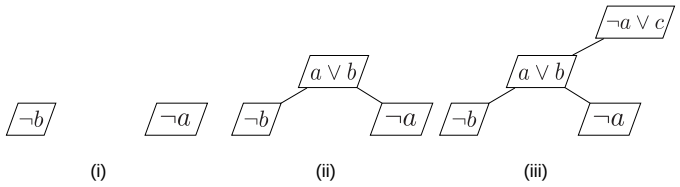

Figure 6.12: Sample Opposition Graphs

Practically, Algorithm 6 can include this by adding the constraint that the opposition sub graph induced by candidate sets of goals must be connex. Connexity constraints can be handled by well-known constraint programming techniques on graph structures [DDD05].

This additional constraint breaks candidate sets of goals into smaller sets. Some smaller set might occur several times, because previously included in several candidate sets of goals, they are considered only once.

Variable opposition graphs can be constructed quickly from the set of goals and connexity constraints on this graph is not expensive to handle [DDD05]. Testing the realizability of the union of two disjoint sets of goals is slower than testing the realizability of these two sets separately because the time complexity of realizability testing is more than linear in term of the number of goals. The technique provides a speedup because the overhead of building the opposition graph and managing the connexity constraint is negligible compared to the gain in the second step of our divergence detection technique. This is validated experimentally in Section 6.6.

6.5.5.3 Making the Detection Incremental

Our divergence detection technique is intended to be applied on successive snapshots of an evolving requirements model. The difference between two successive snapshots is typically that some goals have been modified or added. We only need to search for divergences involving such goals. This enables us to filter out candidate sets of goals that do not include updated goals.

If a candidate set of goals does not include an updated goal, there is no need to test if there is a divergence within this set because this test has already been performed when divergence detection was performed on a previous snapshot. Hence, run time is saved. Put in other words, a set of potentially divergent goals must include at least one updated goal. This

constrain can be added to Algorithm 6, similarly to the previous one.

This constraint might anyway cause the divergence detection to detect divergences that do not involve updated goals. This is because candidate sets of goals including an updated goal might include divergences that do not involve an updated goal.

The sets of all divergences D in the latest snapshot can be computed from the set of divergence D_{new} detected with this incrementality constraint as follows:

$$D = D_{new} \cup (D_{old}/D_{updated})$$

where D_{new} is the sets of divergences incrementally detected on the latest snapshot, D_{old} is the sets of all divergences in the previous snapshot and $D_{updated}$ is the set of divergences in the old snapshot that involve a goal that has been updated since then.

6.5.5.4 Searching for divergences involving fewer goals:

Another technique for making candidate sets of goals smaller and fewer is to look for divergences involving at most n goals where n is decided à priori. This restricts the completeness of our technique in the sense that divergence with more then n goals will not be detected. Such property is similar to the ones of other bounding techniques (bounded model checking [BCC+03], finite instantiation for first-order problems [Jac00]).

This technique might provide either a speedup or a slowdown because bounding the cardinality of candidate sets of goals implies that we increase the number of these sets. For instance, if a candidate set of goals includes three goals, say $\{g_1, g_2, g_3\}$ and we search for divergences involving at most two goals, then we generate three candidate sets: $\{g_1, g_2\}$, $\{g_1, g_3\}$ and $\{g_2, g_3\}$. The realizability-testing algorithm will take less time because applied on fewer goals, but it will be performed three times.

We present a rough numerical estimate of the speedup / slowdown in case there is one candidate set of N goals compared to the same situation where the cardinality is bounded to P. In such a theoretical case, the number of sets obtained by bounding the cardinality is C_N^P. The complexity of realizability testing lies between $O[2^k]$ and $O[2^{2^k}]$ depending on wether or not the tested assertions use the $\Diamond$ operator. The k can roughly be approximated as the number of goals that are tested together for realizability.

Table 6.1 summarizes the speedup in case the complexity is $O[2^k]$ for N and P ranging from 1 to 15. Speedup values are abstracted: an empty cell expresses that the bound is larger than the original cardinality, a "plus" symbol (resp. "minus" symbol) cell expresses that the time consumption is increased (resp. decreased or unchanged) in case the cardinality is bounded.

P \ N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		-	-	-	-	-	-	-	-	-	-	-	-	-	-
2			+	+	+	-	-	-	-	-	-	-	-	-	-
3				+	+	+	+	+	+	-	-	-	-	-	-
4					+	+	+	+	+	+	+	+	+	-	-
5						+	+	+	+	+	+	+	+	+	+
6							+	+	+	+	+	+	+	+	+
7								+	+	+	+	+	+	+	+
8									+	+	+	+	+	+	+
9										+	+	+	+	+	+
10											+	+	+	+	+
11												+	+	+	+
12													+	+	+
13														+	+
14															+
15															

Table 6.1: Estimating the speedup of bounding the cardinality of candidate sets of goals

From Table 6.1, we can see that introducing such a bound does not always introduce a speedup. Speedup tends to appear when the bounding is significantly smaller than the initial cardinality. On the positive side, the bounding is increasingly more attractive if the initial cardinality of potentially divergent goals increases. This result indicates that the bounding might help to make our divergence detection technique scale up to large requirements models.

We have performed the same numerical study in case the complexity is doubly exponential. In this case, there is no "plus" zone. Bounding the cardinality of potentially divergent goals seems to be always more attractive in this case.

The accuracy of such estimate is hindered by the following phenomenons:

- it is based on a ratio of "Big O" notations, which neglects multiplicative constants;
- it assumes that the maximal complexity is always reached, which is (hopefully) not always the case.
- some goal might contribute more than other to the size of the problem, this estimate assumes that they all contribute with similar weights.
- as candidate sets are smaller, like the ones delivered by re-fractioning, they are more likely to be eliminated by the two techniques presented above,

- the overall time spent in computing the candidate sets might take increasingly more time with the number of candidate sets

Practically, the boundary between the "plus" and the "minus" zones in Table 6.1 must be estimated based on a statistical analysis of the efficiency of the final tool and on features of the requirements model. With such table, we can use a threshold technique where candidate sets with cardinality lower than the threshold are not fractioned, while candidate sets with higher cardinality are substituted with all their subsets of a given small cardinality. For instance, according to Table 6.1, if the small cardinality is say 2, the threshold can be set to 6. The technique for choosing candidate sets would be as follows:

- **Step 1:** Search for candidate sets with a cardinality $\geq$ bound;
- **Step 2:** From these sets, discard the ones with a cardinality $\geq$ threshold;
- **Step 3:** Search for sets with cardinality $\leq$ bound that are not subsets of the remaining ones.

The candidate sets are the ones remaining after step 2 plus the ones discovered at step 3.

6.6 Illustration on Prototype Implementation

We have built a proof-of-concept tool that finds divergences in goal models. The tool is named *SSV* (Symbolic Specification Validator). It implements the techniques described in Section 6.5.

The purpose of this prototype is twofold: (1) to validate the refined definition of divergence on examples; (2) to validate our candidate set pruning techniques and assess whether or not they provide some speedup.

This section first presents SSV . It then presents an interactive use of the tool for elaborating a train control system. Finally, run times of the prototype produced on this train control system are analyzed in order to estimate the speedup provided by the candidate set pruning techniques on this example.

6.6.1 SSV : a Prototype for Detecting Divergences

The input of the tool is an `ASCII` file representing a goal-oriented requirements model. It includes two sections. The first one declares the state variables of the system, the second defines the goal model.

The variable declaration section starts with the keyword `variables` followed by the variable names. No type is given because SSV only supports Boolean variables. For instance, in the Mine Pump System, this section is as follows:

```
variables
PumpSwitchOn
HighWater HighWaterSignal
CriticalMethane CriticalMethaneSignal
WaterWarningLampOn
```

The second section may contain two kinds of constructs: goals and refinements. Refinements are included in the input file, so that SSV can organize the detected divergences as root and implied divergences.

Goals are defined by their name, their formal definition and the sets of variable they intent to constraint. An example of goal definition is:

```
goal "Achieve[Pump Switch On When High Water]"
constraints PumpSwitchOn
formaldef HighWater => oo PumpSwitchOn
```

Goal names are used both for reporting the divergences discovered by SSV and for referencing goals in goal refinement declaration.

In order to support incremental divergence detection, we need to represent that a goal has been updated since the previous snapshot. Different input files represent different snapshots. Each file includes the necessary information about its differences with the previous one. Updated goals are prefixed with the keyword `updated`. For instance, an updated goal is defined as follows:

```
updated goal "Achieve[Pump Switch On When High Water Signal]"
constraints PumpSwitchOn
formaldef HighWaterSignal => o PumpSwitchOn
```

Refinements start with the keyword `refinement` followed by the name of the father goal. The names of the children goals are given after the keyword `children`. Every goal referenced in a refinement must be declared prior to the refinement.

```
refinement "Achieve[Pump Switch On When High Water]"
children "Achieve[Pump Switch On When High Water Signal]"
         "Maintain[High Water Signal Accurate]"
```

The output of the tool is the set of divergences in the goal model enforcing the vocabulary disjointness constraint resented in Section 6.5.3. In case the tool is used in an incremental way, the output of the tool only includes divergences involving updated goals. SSV can organize divergences as root and implied divergences.

For each divergence, SSV can produce a boundary trace. The boundary traces delivered by the tool are both optimized in depth and width as explained in Section 6.5.4.3.

When submitted with the MinePump system, SSV produces the output shown in Figure 6.13. Figure 6.14 presents the boundary trace generated by SSV when it is not optimized in size. SSV can produce graphical output representing the goal graph and the divergences it detected, via the graphviz tooktit [EN]. The graphical output produced for the MinePump is shown in Figure 6.15.

```
checking goal file [examples/minepump.goal]
found 1 divergences (and 3 implied divergences)

Divergence (3 implied)
concerned goals:[Achieve[PumpSwitchOnWhenHighWater]
                 Achieve[PumpSwitchOffWhenCriticalMethane]]
How to make them diverge:
     +at time 0
     +output: PumpSwitchOn:any
     +input: CriticalMethane:true HighWater:true
        +at time 1
        +output: PumpSwitchOn:any
        +input: CriticalMethane:false HighWater:false
           +at time 2
           +no possible outputs
```

Figure 6.13: Divergences delivered by SSV on the MinePump

6.6.2 Implementation Details of SSV

SSV is implemented in the Oz programming language featuring built-in constraint programming facilities [RH04]. It uses the BDDLIB BDD library for representing sets of states and transition relations [Bry86, Lon]. This library provides all the necessary features.

SSV supports the three techniques presented in Section 6.5.5 for optimizing the generation of candidate sets of goals. They can be activated or deactivated via command-line arguments.

```
How to make them diverge:
     +at time 0
     +output: PumpSwitchOn:true
     +input: CriticalMethane:true HighWater:true
     |  +at time 1
     |  +output: PumpSwitchOn:true
     |  +input: CriticalMethane:false HighWater:false
     |  |  +at time 2
     |  |  +no possible outputs
     |  +output: PumpSwitchOn:false
     |  +input: CriticalMethane:false HighWater:false
     |     +at time 2
     |     +no possible outputs
     +output: PumpSwitchOn:false
     +input: CriticalMethane:true HighWater:true
        +at time 1
        +output: PumpSwitchOn:true
        +input: CriticalMethane:false HighWater:false
        |  +at time 2
        |  +no possible outputs
        +output: PumpSwitchOn:false
        +input: CriticalMethane:false HighWater:false
           +at time 2
           +no possible outputs
```

Figure 6.14: Non-simplified boundary trace for the MinePump

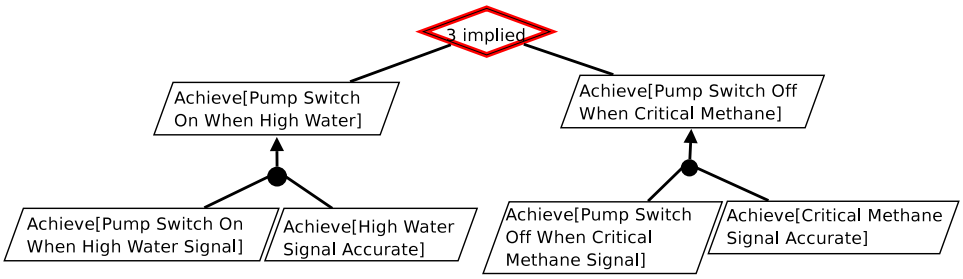

Figure 6.15: Graphical output of SSV for the MinePump

The incremental divergence detection is however not entirely supported. The divergence included in a requirements model can be computed incrementally by combining the new divergences found at a given snapshot with the one known at the previous snapshot as discussed in Section 6.5.5.3. SSV does not support this combination operation, because it is only given a snapshot of the requirements model, not its output when executed on the previous snapshot. This issue has to be solved when integrating SSV within an interactive application for elaborating goal-oriented requirements model because such framework would support the storage of divergences and would therefore be able to combine the new divergences with the ones known at the previous snapshot [obj].

The candidate set bounding technique is implemented through a pair of numbers (threshhold,bound) provided at the command line as described in Section 6.5.5.4.

SSV is partially unsound because it does not perform determinization of the intermediary transition system it manipulates for testing realizability. We therefore need to manually tune the formal definition of goals so that the transition system generated inside SSV is deterministic right from the start. The determinization procedure, through well-known turned out to be too costly to implement because the transition system manipulated by SSV is not represented explicitly in memory. In practice, the boundary trace generation procedure fails when it tries to generate the boundary trace for a detected divergence that is not actually a divergence.

6.6.3 Analysing a train control system with SSV

This section shows how SSV can be used for elaborating a train control system The requirements model of the train is gradually enriched either by refining goal, correcting divergences or adding new goals and SSV is run The result of SSV is used to drive the requirements elaboration process.

The graphical output of SSV is reported for each snapshot. For clarity, SSV draws updated goals with blue lines.

Within this section, the incremental divergence detection is deactivated in order to present all the divergences that can be found by SSV .

Computation times are also reported for each snapshot. These are measured with incremental divergence detection activated because they are close to the actual time a user would wait when using SSV . Timing measurements have been performed on a Pentium Centrino Duo @ 1.66GHz (only one core was used here because SSV is single-threaded). The measured run times do not include the computation of boundary trace. the issue of whether these run time are accurate measures of performances that can be expected from

a real tool are discussed in Section 6.6.4.

The initial snapshot only includes the two following goals:

Goal Maintain[Door Closed While Moving]
Constrained variables: $DoorsOpen$
Formal Def $Moving \Rightarrow \neg DoorsOpen$

Goal Achieve[Doors Open When Fire]
Constrained variables: $DoorsOpen$
Formal Def $Fire \Rightarrow \Diamond_{\leq 2} DoorsOpen$

These two goals are defined on the domain made of the three variables:

- Fire = "there is fore onboard"
- Moving = "the train is moving"
- DoorsOpen = "the doors of the train are open"

These goals are divergent because they cannot be enforced when there is fire and the train keeps one moving for two time units. This is confirmed by SSV in Figure 6.16 (computation time: 10ms). The boundary trace delivered by SSV for this divergence is shown in Figure 6.17.

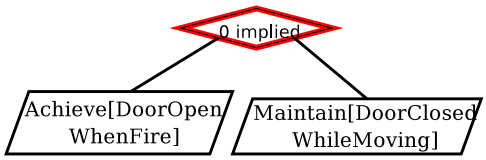

Figure 6.16: First snapshot of train control system

This is resolved by prescribing that the train should stop when there is fire onboard. This gives rise to the following new goal:

Goal Achieve[Train Stopped When Fire]
Constrained variables: $Moving$
Formal Def $Fire \Rightarrow \circ\neg Moving$

This is the boundary trace avoidance tactics [vLDL98]. SSV cannot detect that there is a boundary trace avoidance goal in a requirements model because our technique for detecting divergences does not support domain properties; boundary condition avoidances should be handled like a domain property. In order to avoid having SSV reporting on this divergence, we mention that the goal *Achieve[Doors Open When Fire]* constraints the variable *Moving* too.

```
+at time 0
+output: DoorsOpen:false
+input: Fire:true Moving:false
|  +at time 1
|  +output: DoorsOpen:any
|  +input: Fire:false Moving:true
|     +at time 2
|     +output: DoorsOpen:true
|     +input: Fire:false Moving:true
|        +at time 3
|        +no possible outputs
+output: DoorsOpen:true
+input: Fire:false Moving:true
   +at time 1
   +no possible outputs
```

Figure 6.17: Boundary trace of the divergence between *Maintain[Door Closed While Moving]* and *Achieve[Doors Open When Fire]*

Goal Achieve[Doors Open When Fire]
Constrained variables: DoorsOpenMoving
Formal Def Fire $\Rightarrow \Diamond_{\leq 2}$DoorsOpen

The resulting model is shown in Figure 6.18 (computation time: 10ms).

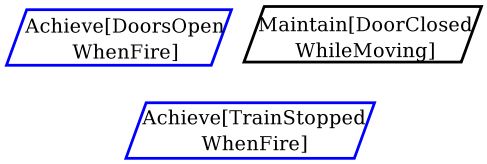

Figure 6.18: Second snapshot of train control system

The goal *Achieve[Train Stopped When Fire]* is actually part of the refinement of the goal *Achieve[Doors Open When Fire]*. We therefore complete the refinement of this goal with the two following new goals:

Goal Achieve[Doors Open When Stopped Because Of Fire]
Constrained variables:*DoorsOpen*
Formal Def$\blacklozenge_{\leq 1} Fire \wedge \bullet\neg Moving \Rightarrow DoorsOpen$

Goal Avoid[Train Restarts Without Opening Doors]
Constrained variables: $Moving$
Formal Def $\blacklozenge_{\leq 1} Fire \wedge \neg Moving \Rightarrow (\neg Moving)\ \mathcal{W}\ \neg DoorsOpen$

The resulting snapshot is shown in Figure 6.19 (computation time: 30ms).

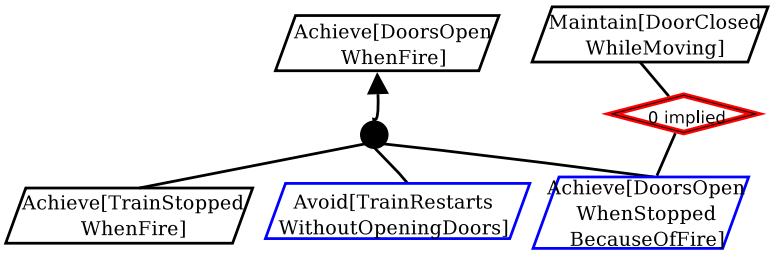

Figure 6.19: Third snapshot of train control system

There is a new divergence. The boundary trace is given in Figure 6.20. On this boundary, we see that the train starts after having stopped because of a fire. This is again avoided by a boundary condition avoidance goal. In this case, it is the goal *Avoid[Train Restarts Without Opening Doors]*

```
+at time 0
+output: DoorsOpen:false
+input: Fire:true Moving:false
|  +at time 1
|  +output: DoorsOpen:true
|  +input: Fire:false Moving:true
|     +at time 2
|     +no possible outputs
+output: DoorsOpen:true
+input: Fire:false Moving:true
   +at time 1
   +no possible outputs
```

Figure 6.20: Boundary trace of the divergence between *Achieve[Doors Open When Stopped Because Of Fire]* and *Maintain[Door Closed While Moving]*

We now extend the requirements model with a functional goal prescribing that the train should move when the tracks are clear. This involves a new state variable *ClearTracks* denoting that the tracks are clear.

Goal Achieve[Train Progress When Clear Tracks]
Constrained variables:$Moving$
Formal Def$ClearTracks \Rightarrow \circ Moving$

The resulting snapshot is shown in Figure 6.21 (computation time: 20ms). Unsurprisingly, the new goal is divergent with the goal *Achieve[Train Stopped When Fire]*.

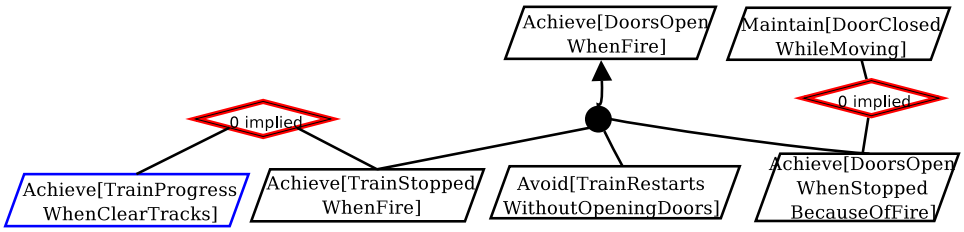

Figure 6.21: Fourth snapshot of train control system

Instead of correcting the new divergence immediately, we refine the functional goal. In order to see that there are some implied divergence appearing.

The train driver is unable to monitor whether the tracks are clear or not. Therefore, we introduce an additional variable *GoSignal* to resolve this lack of monitorability. This gives rise to the two following new goals:

Goal Achieve[GoSignalSendWhenClearTracks]
Constrained variables:$GoSignal$
Formal Def$ClearTracks \Rightarrow GoSignal$

Goal Achieve[TrainProgressWhenGoSignal]
Constrained variables:$Moving$
Formal Def$GoSignal \Rightarrow \circ Moving$

These goals refine *Achieve[TrainProgressWhenClearTracks]*. The resulting snapshot is shown in Figure 6.22 (computation time: 30 ms). The divergence involving *Achieve[Train Progress When Clear Tracks]* has now one implied divergence.

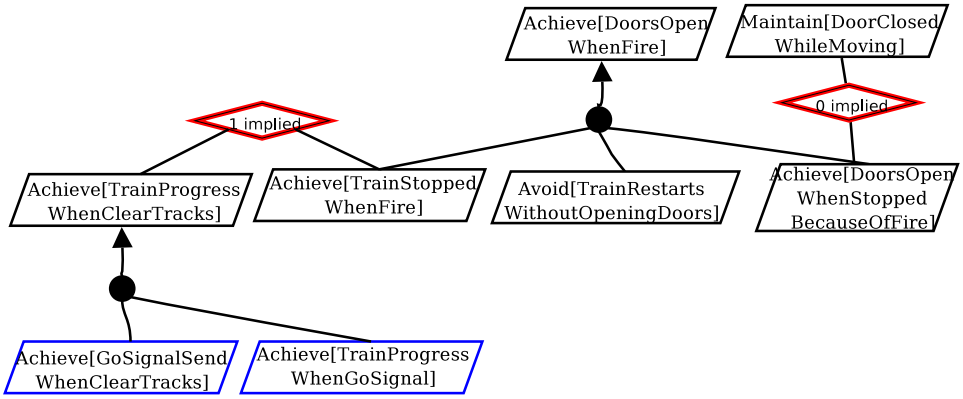

Figure 6.22: Fifth snapshot of train control system

We now focus on correcting these divergences. The root divergence is resolved by weakening *Achieve[Train Progress When Clear Tracks]* into the following:

Goal Achieve[Train ProgressWhenClearTracksAndNoFire]
Constrained variables: $Moving$
Formal Def $ClearTracks \wedge \blacksquare_{\leq 2} \neg Fire \Rightarrow \circ Moving$

The resulting snapshot is shown in Figure 6.23 (computation time: 30 ms).

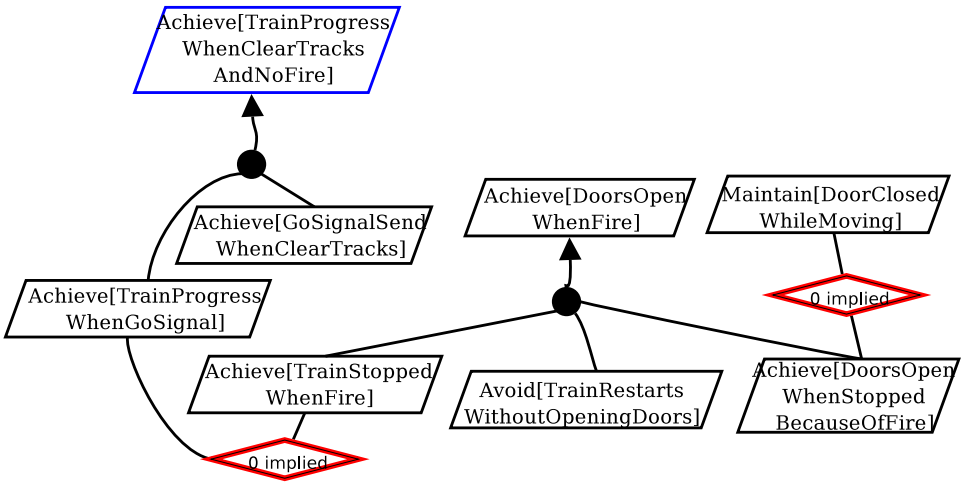

Figure 6.23: Sixth snapshot of train control system

This resolves the root divergence. The divergence that was previously implied becomes a root divergence.

The new root divergence is resolved similarly by propagating the weakening to *Achieve[Train Progress When Go Signal]*. the weakened goal is as follows:

Goal Achieve[Train Progress When Go Signal And No Fire]
Constrained variables: $Moving$
Formal Def $GoSignal \wedge \blacksquare_{\leq 2} \neg Fire \Rightarrow \circ Moving$

We also introduce a goal prescribing that the train should not move if the driver is not responsive. This introduces a new variable *DriverResponsive*. this variable represents a button that the driver should press regularly (i.e.: at least once every three time units) in order to prove to the train control system that he is able to drive the train. The new goal is defined as follows:

Goal Avoid[TrainMovingWhenDriverNotResponsive]
Constrained variables: $Moving$
Formal Def $\blacksquare < 3 \neg DriverResponsive \Rightarrow \circ \neg Moving$

The resulting snapshot is shown in Figure 6.24 (computation time: 70ms).

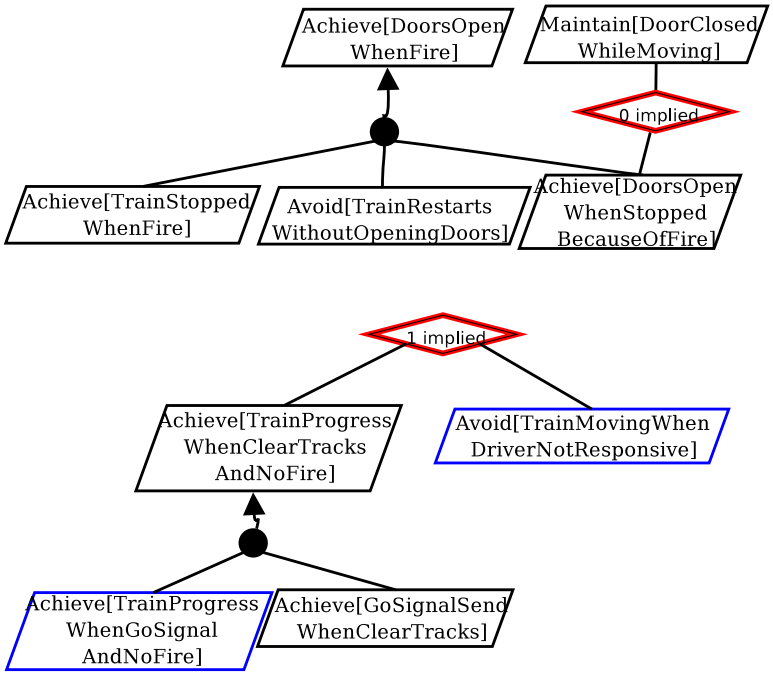

Figure 6.24: Seventh snapshot of train control system

The new goal is divergent with functional goals prescribing that the train should move. We leave these divergences unresolved for now. We rather focus on a new safety goal prescribing that doors should be closed when the train is not at a station. The domain is extended with a new variable *AtStation* denoting that the train is at the station. The new goal is defined as follows:

Goal Maintain[Door Closed Between Station]
Constrained variables: *DoorsOpen*
Formal Def $DoorsOpen \Rightarrow AtStation$

The resulting snapshot is shown in Figure 6.25 (computation time: 30ms).

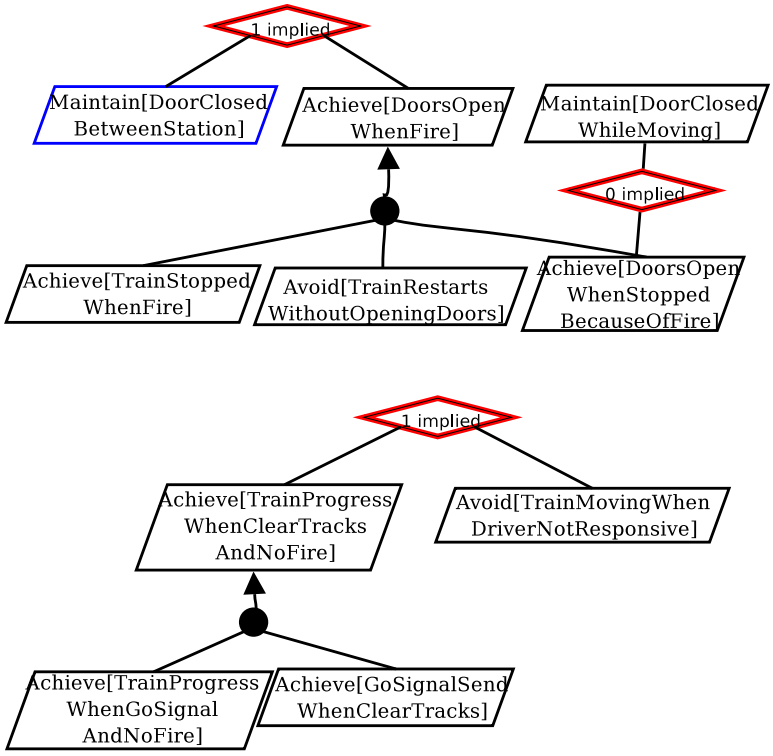

Figure 6.25: Eight snapshot of train control system

The new goal is divergent with the goal *Achieve[Doors Open When Fire]*. There is one implied divergence.

We resolve the divergence involving the safety goal *Maintain[Door Closed Between Station]* by weakening it into *Maintain[Door Closed Between Station when No Fire]*.

Such resolution must be carefully discussed with expert of the domain. We assume here that this is the adequate resolution. The weakened goal is as follows:

Goal Maintain[Door Closed Between Station When No Fire]
Constrained variables: $DoorsOpen$
Formal Def $DoorsOpen \Rightarrow AtStation \vee \blacklozenge_{\leq 2} Fire$

The resulting snapshot is shown in Figure 6.26 (computation time: 40ms).

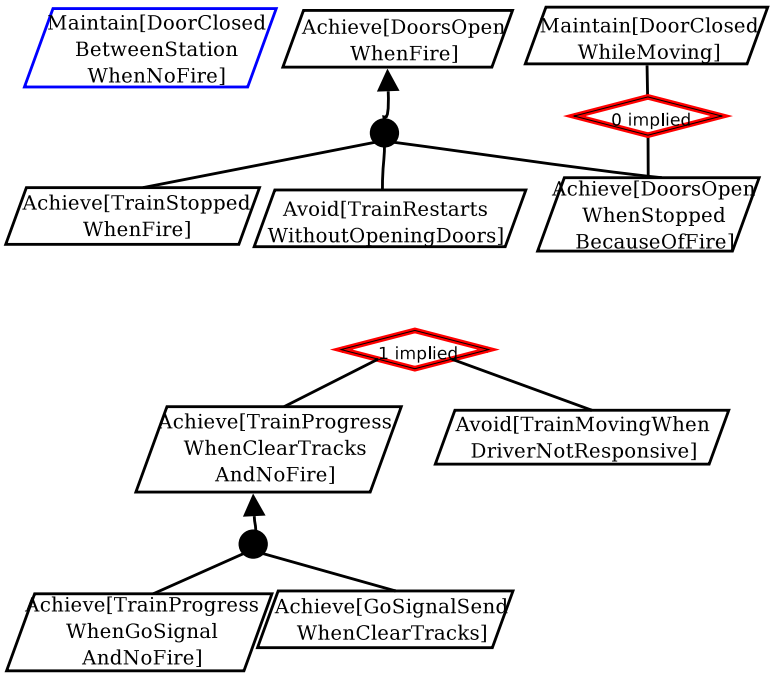

Figure 6.26: Ninth snapshot of train control system

We now focus on resolving the divergence involving *Achieve[Train Progress When Clear Tracks And No Fire]*. This divergence is resolved by weakening this functional goal as follows:

Goal Achieve[Train Progress When Clear Tracks And No Fire And Driver Responsive]
Constrained variables: $Moving$
Formal Def
$ClearTracks \wedge \blacksquare_{\leq 2} \neg Fire \wedge \blacklozenge_{<3} DriverResponsive \Rightarrow \circ Moving$

This resolves the divergence and discloses the previously implied one. The resulting snapshot is shown in Figure 6.27 (computation time: 130 ms).

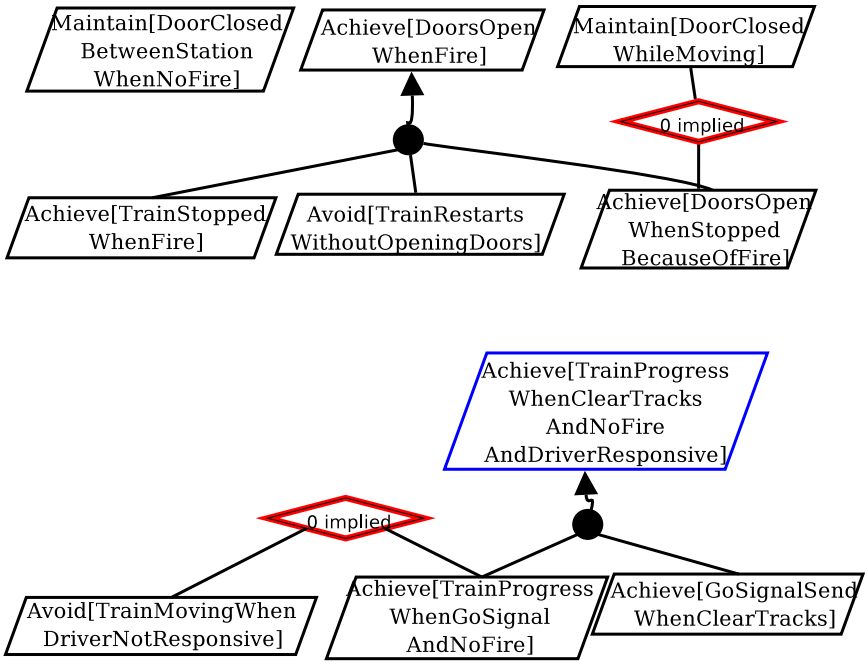

Figure 6.27: Tenth snapshot of train control system

We propagate the weakening to the new root divergence. the goal *Achieve[Train Progress When Go Signal And No Fire]* in weakened into the new goal *Achieve[Train Progress When Go Signal And No Fire And Driver Responsive]* defined as follows:

Goal Achieve[Train Progress When Go Signal And No Fire And Driver Responsive]
Constrained variables: $Moving$
Formal Def
$GoSignal \wedge \blacksquare_{\leq 3} \neg Fire \wedge \blacklozenge_{<3} DriverResponsive \Rightarrow \circ Moving$

This resolves the divergence. The resulting snapshot is shown in Figure 6.28 (computation time: 190 ms).

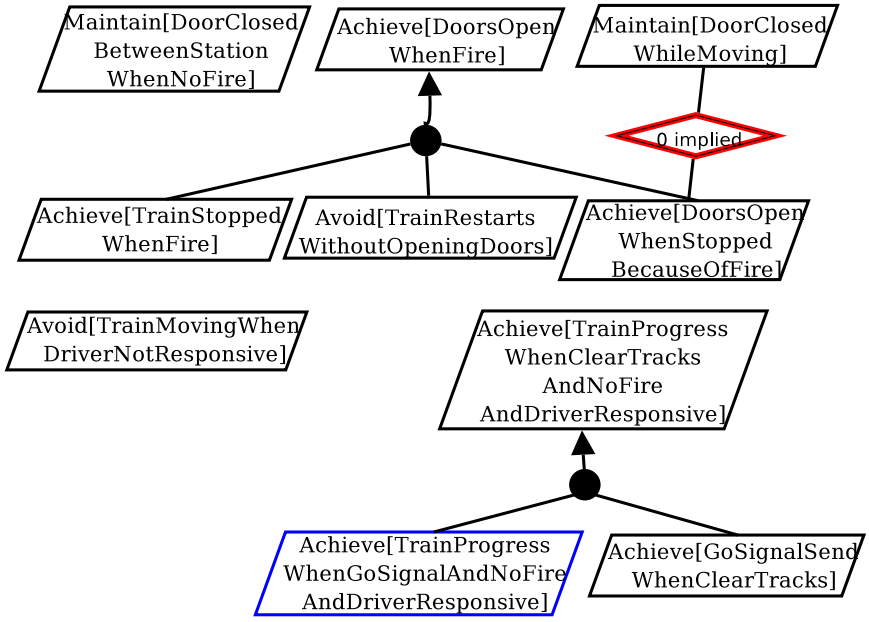

Figure 6.28: Eleventh snapshot of train control system

At this stage, we focus on the goal *Maintain[Door Closed While Moving]*. This goal was said to constraint only the variable *DoorClosed*. We to enforce this goal by ensuring that the train waits for doors to be closed before moving. We can therefore state that this goal also constraints the variable *Moving*. This removes the spurious divergence involving this goal. The updated definition is as follows:

Goal Maintain[DoorClosedWhileMoving]
Constrained variables: $DoorsOpenMoving$
Formal Def $Moving \Rightarrow \neg DoorsOpen$

This goal is refined by requiring that a moving train cannot open its door, and by requiring that a stopped train must close its door before starting. An additional goal is needed to prescribe that a train does not open its door when starting at the same time. The new goals are defined below. They constitute the refinement of *Maintain[Door Closed While Moving]*.

Goal Avoid[Start When Doors Open]
Constrained variables: $Moving$
Formal Def $\neg Moving \wedge DoorsOpen \Rightarrow \circ\neg Moving$

Goal Avoid[Open Doors When Moving]
Constrained variables: $DoorsOpen$
Formal Def $Moving \Rightarrow \circ\neg DoorsOpen$

Goal Avoid[Open Doors When Starting]
Constrained variables: $DoorsOpen$
Formal Def $\neg Moving \wedge \circ Moving \Rightarrow \circ\neg DoorsOpen$

The resulting snapshot is shown in Figure 6.29 (computation time: 5350 ms). For this snapshot, only the relevant concepts are represented in order to keep the figure light.

We can see that there are two ternary divergences involving train progress and door closed.

Our analysis stops here because we have shown a typical use of SSV and the following steps of the requirements elaboration are similar to the ones already presented in this section.

6.6.4 Efficiency measurements

This section presents the efficiency measurements of the prototype on the train case study. The purpose of these measurements is to check that the techniques proposed in Section 6.5.5 decrease the overall run time as they

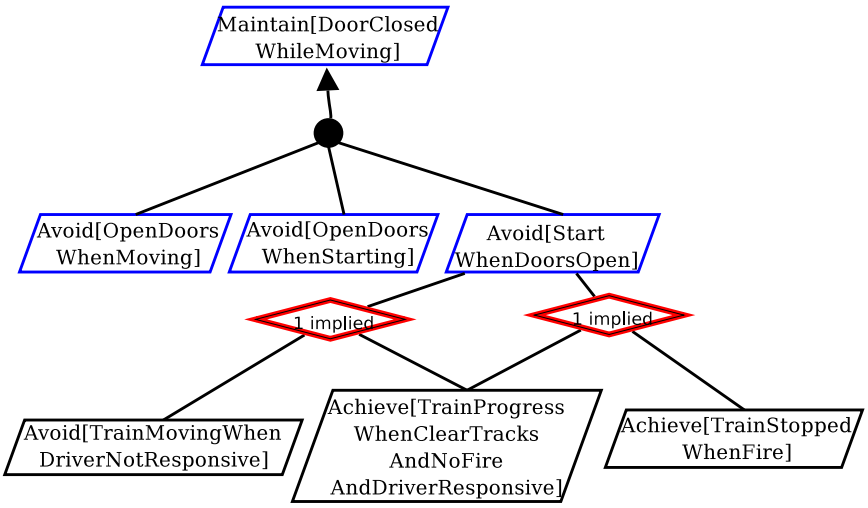

Figure 6.29: Twelfth snapshot of the train control system.

are intended to.

On Table 6.2, we have reported the cardinalities of candidate sets of goals for each snapshot of the train control system, with various combinations of optimizations. Rows represent the snapshots of the train control system. Columns represent the candidate sets optimization technique selected. Cells contain the cardinalities of the candidate sets of goals for the snapshot of the row, with the optimizations of the column.

Snapshots	No optimization	Incremental	Connex	Connex & Incremental
1	2	2	2	2
2	2 1	2	1 1 1	1 1
3	2 2 2	2 2	1 2 2	2 2
4	3 2 3	3 3	1 2 3	3
5	4 3 4 4 4	4 3 4 4 4	1 2 4 1	1 4
6	4 3 4 4 4	4 4 4 4	1 2 4 1	4
7	5 3 5 5 5	5 5 5 5	1 2 5 1	5
8	6 4 6 5 5	6 6 4	2 3 5 1	2 3
9	6 4 6 5 5	6 6 4	2 3 5 1	2 3
10	6 4 6 5 5	6 6 5 5	5 2 3 1	5
11	6 6 4 5 5	6 5	5 2 3 1	5
12	8 8 4 6 6	8 8 4 6 6	8 3 6 1	8 3 6

Table 6.2: Reporting the cardinalities of candidate sets

On Table 6.2, we clearly see the impact of the optimizations. The *incremental* optimization alone allows dropping some of the candidate sets.

The *connex* optimization alone allows breaking candidate sets of goals into smaller sets. There might be several occurrences of the same sets. These are considered only once. This leads to a lowering of the number of candidate sets. From these smaller sets, some of the remaining sets are dropped when the "incremental" constraint is added to the "connex" constraint.

Our two techniques perform poorly on the last snapshot, where they cannot break a candidate set of size eight. This is because at this snapshot, we have updated a lot of goals and these goals reference many different variables of the system, thus the variable opposition graph is much more connected.

We now report on timing measures performed on our prototype. Timing measurements have been performed on a Pentium Centrino Duo @ 1.66GHz (only one core was used here because SSV is single-threaded). The measured run times do not include the computation of boundary trace. These timing measures must be taken with care because the algorithm spends a non-negligible amount of time encoding goals into SNF and encoding SNF into BDD's. Goals can be conjoined once they are encoded into BDD's. The BDD encoding of goals can be cached if the tool were made interactive.

Figure 6.30 reports the absolute run time of our prototype on the snapshots of the train control system. Run times have been measured with incremental divergence detection and connexity constraint.

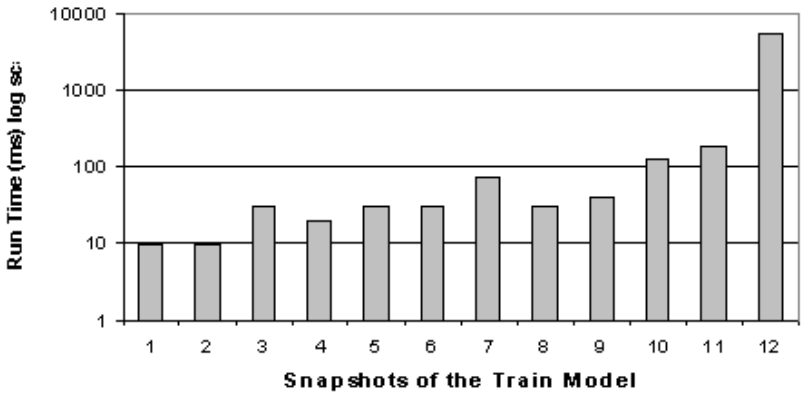

Figure 6.30: Run time of divergence detection on the Train control system

The timings are reasonably low on such little example except for the last snapshot, where the cardinality candidate set grow to eight goals and our homebrew realizability testing technique performs poorly. The good news is that the time does not strictly increase with the snapshot number as for snapshots 4, 5, 6, 8, and 9. This is mostly due to the incrementality of the

detection.

The speedups provided by the incrementality and the connexity constraints compared to the basic constraint alone are reported in Figure 6.31. The represented numbers are the speedups rounded to the nearest integer.

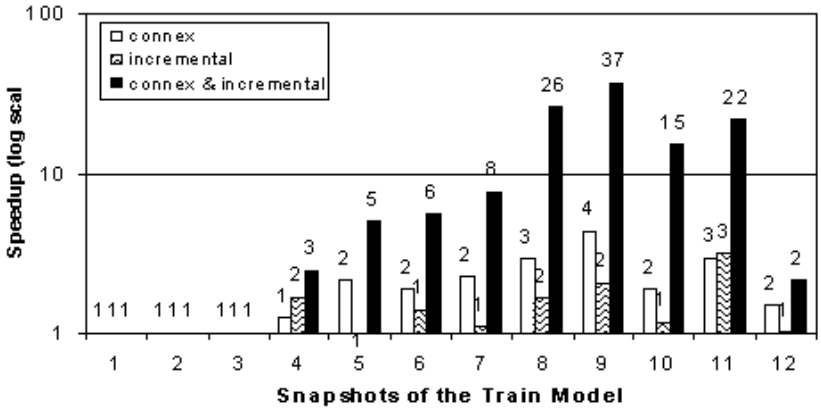

Figure 6.31: Measuring the speedup provided by enhanced pruning techniques on the overall divergence detection process

Figure 6.31 clearly shows that the two concerned techniques provide a speedup on this example. The overall speedup is not directly related to the candidate sets number and cardinalities because of the processing overhead due to these techniques themselves. The speedup is unfortunately very low for the last snapshot. This might have been expected from the cardinalities of candidate sets in Figure 6.2.

The technique for setting an arbitrary bound to the cardinalities of candidate sets described in Section 6.5.5.4 does not provide any relevant speedup when applied on this case study. This is because the cardinalities of the candidate sets fall in the categories where it does not work if the bound is two or more.

6.7 Related Work

The words "inconsistency detection" have been given numerous meanings across software engineering disciplines. The most encountered meanings in requirements engineering include [vLDL98]

- **Terminology clashes** that occur when a single real-world concept is given different syntactic names in the requirements specification. For instance, in the e-signature system, we may say that a signer is at a PC

or that a signer is the user of the PC, leading to different predicates for denoting the same concept.

- **Designation clashes** that occur when a single syntactic name in the requirements specification designates different real-world concepts. For instance, in the e-signature system, we may say that a the user of a PC is either the person sitting at the PC or the one that has legitimate login on rights on it.

- **Structure clashes** that occur when a single real-world concept is given different structures in the requirements specification. For instance, we may say that the SignBox is a device connected to a PC or that it is a bundle made of a PC plus an external e-purse reader device.

- **Conflicts or divergences** that occur when there are some circumstances that require an action to be taken and not taken at the same time. This is the type of inconsistencies discussed throughout this chapter. These might occur between requirements or high-level goals.

- **Obstacles** that are special cases of the previous ones where a requirement conflicts with a domain property, as presented in Section 3.2.5.4.

Throughout this chapter, we exclusively focused on conflicts and divergences. Conflicts are special cases of divergences where the boundary condition is tautological. Conflict and divergence detection techniques can be organized into the following taxonomy [SG01]:

- **theorem-proving-like approaches** characterized by the use of some formal inference technique to symbolically derive inconsistencies from software models expressed in a formal modeling language. The technique for detecting divergences presented in [vLDL98] belongs to this category. The principle is to symbolically derive a boundary condition by backward-chaining goals, expressed in first order temporal logics through a domain theory. The problem with these techniques is that they are generally semi-decidable and they only solve the divergence-testing problem. On the other hand, such techniques generate a boundary condition.

- **model-checking approaches** are characterized by the use of a model checker for detecting inconsistencies.

  These techniques mainly apply to requirements engineering technique that support state-machine models, such as the SCR and RSML methods [HJL96, CH02]. In these methods, there might be inconsistencies

between given declarative requirements and a given operational state-machines represented through tabular notations. Model-checking approaches do not guarantee that there exists a state machine implementing a given set of requirements, which is the problem discussed throughout this chapter.

Such techniques can also be used in order to detect divergences, in the sense of the definition presented in Section 3.2.5.3. In this approach, the principle was to search for a trace T suggesting a boundary condition for a divergence involving goals $\{G_i\}_i$ within domain *Dom* by solving the following problem:

$$T \models \text{Dom} \wedge \neg \wedge_i G_i$$

This trace might either suggest a boundary condition or be irrelevant, as this problem does not refer to the realizability of the boundary condition by the environment. This approach is therefore unsound, but it is much more tractable than the one proposed in this chapter.

- **specialized automated analysis approaches** are characterized by the use of any formal tool for detecting one of the inconsistency defined here.

  A technique for detecting conflicts between requirements has been proposed in [GZ05]. This technique translates natural language requirements into non-temporal propositional logics. The latter is formally analyzed for inconsistencies. This technique is restricted to the inconsistencies between requirements to non-temporal propositional requirements. Our technique integrates temporal aspect and enables the anticipation of inconsistencies at requirements level when only high-level goals are identified. We also provide technique for illustrating inconsistencies through tree-structured traces satisfying the boundary conditions.

- **human-centered collaborative exploration approaches** have been proposed in numerous requirements engineering techniques where goals and requirements are not formally defined [MZ06, BPG$^+$04, GMMZ06, CNYM00, Yu97]. Common patterns of conflicting goals have been proposed in order to easily identify potentially conflicting goals [vLDL98]. Our technique for pruning the candidate sets of goals based on their vocabulary (see Section 6.5.5.2) can be seen as a specialization of this technique.

6.8 Conclusion

This chapter proposed an automated technique for detecting divergences. The principle is to generate sets of goals taken from the goal model on which a realizability-testing algorithm can act as a divergence tester. Such sets of goals are called candidate sets of goals.

In order to avoid testing the realizability of every combination of goals taken from the goal model, we first proposed a customized realizability-testing technique that computes all the minimal sets of goals included in a candidate set such that they are not realizable. These are the divergent goals included in the candidate set of goal. This modified technique requires that sets of goals submitted to it match some property on the vocabulary of goals. The key idea of our divergence detection process is to choose sets of goals that are the maximal sets of goals enforcing this property. The sets of goals are then tested for divergence by applying our modified realizability-testing algorithm. Our modified realizability-testing technique delivers all the divergences included in this set.

A technique has been proposed to illustrate divergences by means of tree-structured traces. Three improvements to the realizability-testing procedure have been proposed to make it faster. This is a critical issue because the realizability-testing problem is known to be very hard [PR89b, PR89a, PR90, PPS06]. We focused on techniques to make the candidate sets of goals smaller and fewer because the realizability-testing problem is already subject to intensive research.

A quick vocabulary-based approximation of divergent goals enables to estimate whether goals might or are surely not divergent, based on a lightweight syntactic analysis of their definition. This element of knowledge enables us to split up candidate sets into smaller sets until within each candidate sets, there might be a divergence involving every goal. Incremental divergence detection relies on the fact that the divergence detection process will be performed on successive snapshots of an evolving requirements model. The difference between two successive snapshots is typically that some (typically few) goals have been updated, that is, modified or added. In this situation, we only need to search for divergences involving such updated goals. This enabled us to filter out candidate sets of goal that do not include one such updated goal.

We proposed to bound the cardinality of candidate set of goals in order to get an overall speedup. A numerical estimate indicated that this would indeed provide a speedup for goal graphs with a sufficiently high number of goals. Of course, these techniques make it impossible to detect divergences involving more goals than the given bound. There is a tradeoff between the completeness of the divergence detection and its efficiency.

A prototype tool has been implemented to validate both the new definition of divergence and the three proposed optimizations. It was successfully tested on the requirements model of a small train control system.

6.9 Open Issues

The major issue is that detecting divergences has an unaffordable time complexity. This section presents several tracks towards a scalable solution for detecting divergences.

The bottleneck of our technique is the realizability testing because it is a very expensive problem to solve with a double-exponential time complexity. This case appears in case the tested temporal assertion includes an $\Diamond$ operator. Recent advances have been done and are still ongoing in the field of realizability testing and more efficient techniques are now available for testing the realizability of temporal assertions [WDR06]. An obvious direction for making our technique faster is to use these techniques instead of the simple technique described in Section 6.5.4.1.

Alternative techniques for testing the realizability can be proposed based on approximations such as bounded verification and abstract verification [Jac06, BCC+03, CC77]. These techniques are potentially faster than the exact realizability test discussed here above, but with soundness/completeness trade off. These techniques might be used in conjunction with exact techniques, in order to filter out as many realizability test problems as possible and only apply the exact techniques on the realizability test problems that cannot be solved by these approximate techniques.

Our technique is still limited to domain-independent divergences. Some divergences might not be problematic because the boundary condition cannot be met. This might be due to domain properties or to additional goals that prevent the boundary condition from occurring. These are known as *boundary condition avoidance*. For instance, in a world where there is never water and methane at the same time in a mine, the goals *Achieve[Pump On When High Water]* and *Achieve[Pump Off When Critical Methane]* will never appear to be inconsistent. Introducing boundary condition avoidance goals is a classical way of resolving divergences [vLDL98]. We should therefore add support in our tool for testing that a given assertion is a boundary condition avoidance for a given divergence.

Domain properties can also give rise to additional divergences. For instance, suppose that there is a domain property stating that the pump cannot be turned on for more than five time units. In this case, the goal *Achieve[Pump On When High Water]* is divergent with itself within this domain because there can be water for more than five time units. Divergences

involving one goal and possibly domain properties are known as *obstacles* (see Section 3.2.5.4) [vLL00].

In order to support domain properties, we need to implement a domain-specific realizability-test, and to add some domain properties to candidate sets of goals. Deciding which domain properties should be considered as potential boundary condition avoidance or as divergence inducers should be decided based on their vocabulary and based on the set of constrained variables appearing in the goals.

We did not address this issue here because our realizability-testing procedure is only a partial implementation of the realizability-testing procedure presented in Section 6.5.4.1 as it does not perform a determinization of the intermediate state machine. Our implementation might crash if the intermediate state machine does feature some non-determinism that is resolved by this determinization procedure. The point is that testing the realizability of an assertion within a domain always features such non-determinism. Testing the realizability of an assertion "A" within a domain "Dom" amounts to testing the realizability of assertion "$Dom \rightarrow A$". The implication introduces the unsupported non-determinism.

Obstacle analysis can be seen as a degenerate case of conflict analysis; an obstacle amounts to a condition for conflict between N goals within the domain under consideration, where N=1 [vLL00, vLDL98]. This chapter proposed a technique for detecting divergence. This technique is likely to be applicable to the detection of obstacle. Obstacles generally involve domain properties, so that extending our technique to obstacle analysis requires that it supports domain properties at first [vLL00].

A very important direction for future work is to extend our technique to first-order logics. This would enable us to support goals formally defined in temporal first order logics as presented in Section 6.1. Detecting inconsistencies involving goals specified in first order logics is a non-decidable problem, because it amounts to automated proving of first order theorems. Nevertheless, the problem of detecting divergences between first order temporal assertions can be reduced into detecting divergences between propositional temporal assertions. This can be achieved by bounding the number of instances for entity classes and searching for divergences involving these instances only [Jac06]. This technique is used in the context of the technique for checking confidentiality goals for violation in Section 4.5.2.3.

Another similar open issue is to extend our technique in order to support richer types of variables, including integers, non-interpreted types, enumerations, etc. Typically, this can be implemented as a two-step approach. The first step identifies atomic propositions from the richer assertions. For instance, comparison of integer values can be considered as an atomic propo-

sition. A dedicated technique (such as CP) might identify the relationship between these various propositions. For instance, $a \geq b$,$a = 5$,and $b = 6$ can be taken as a propositions, say P, Q, R, respectively. In this case, these three propositions cannot be true at the same time. These atomic propositions can substitute for the sub-formula they represent in the richer assertions. The latter can be pre-fixed with the relationship between the various propositions. This gives rise to an equivalent model including only propositional variables, on which the techniques presented in this chapter can be applied. This is the second step.

The technique presented in this chapter searches for all divergences appearing in goal models. This turns out to be expensive. An interesting direction for gaining some speedup is to search for *root* divergences only. This might be integrated as an additional constraint in the technique for choosing candidate sets of goals presented in Section 6.5.5.

The technique presented here for anticipating inconsistencies requires annotating goals with constrained variables. This might be considered as a burden. We should examine how to remove the need for this annotation. A first track is to automatically annotate goals with constrained variables, for instance by propagating information from a goal to its children goals or by propagating information about the identified agents to the high-level goals. Another track for removing this annotation is to use additional variables in the realizability test problem for capturing which are the controlled and monitored variables. Similar techniques have been proposed in Section 4.5.2.5.

Unlike the technique presented in [vLDL98] for testing goals for divergence, our technique does not generate a boundary condition. Yet, our technique can generate boundary traces satisfying this boundary condition. Such boundary traces are actually boundary conditions but not weakest boundary condition in the sense defined in Section 6.4.6. For instance, consider the boundary trace shown in Figure 6.10. This boundary trace is equivalent to the following condition:

$$\text{CriticalMethane} \wedge \text{HighWater} \wedge \circ\neg\text{CriticalMethane} \wedge \circ\neg\text{HighWater} \quad (6.8)$$

This condition is an acceptable boundary condition, according to our definition of divergence. The weakest boundary condition is as folllows:

$$\Diamond(\text{CriticalMethane} \wedge \text{HighWater})$$

Future work can be invested in order to generate more general boundary conditions. Our technique for making boundary traces narrow is a first step in this direction. For instance, consider the trace of Figure 6.8. This trace is equivalent to the following condition:

$$\text{PumpOn} \rightarrow$$

$$\left(\begin{array}{l}\text{CriticalMethane} \wedge \text{HighWater} \\ \wedge \circ \left(\begin{array}{l}\text{PumpOn} \rightarrow (\neg\text{CriticalMethane} \wedge \neg\text{HighWater}) \\ \wedge\neg\text{PumpOn} \rightarrow (\neg\text{CriticalMethane} \wedge \neg\text{HighWater})\end{array}\right)\end{array}\right)$$
$$\wedge\neg\text{PumpOn} \rightarrow$$
$$\left(\begin{array}{l}\text{CriticalMethane} \wedge \text{HighWater} \\ \wedge \circ \left(\begin{array}{l}\text{PumpOn} \rightarrow (\neg\text{CriticalMethane} \wedge \neg\text{HighWater}) \\ \wedge\neg\text{PumpOn} \rightarrow (\neg\text{CriticalMethane} \wedge \neg\text{HighWater})\end{array}\right)\end{array}\right)$$

This condition is also an acceptable boundary condition for the considered divergence. It is equivalent to (6.8), but includes some simplifications, thanks to our technique for making boundary traces narrow.

Chapter 7

Conclusion

Elaborating the requirements model is a crucial step in the development of security-critical systems. All development steps coming after the requirements engineering step (that is: architecture elaboration, coding, testing, etc) head towards the implementation of the identified requirements. A missing, incomplete or not enforced security requirement gives rise to a vulnerability in the running system.

Vulnerabilities in the requirements model notably arise because of incompleteness and inconsistencies. Incompleteness refers to missing requirements, inconsistencies refers to requirements prescribing incompatible behaviors.

This thesis proposed three techniques for assisting requirements engineer in the elaboration of requirements models such that the running system will not have vulnerabilities originating from it. We focused on completeness and consistency issues. The first part of this thesis, that is: Chapter 4 and Chapter 5, focused on completeness issues. The second part of this thesis, that is: Chapter 6, focused on consistency issues.

7.1 What Makes Security-Critical Systems so Different?

The engineering of requirements for security-critical systems raises specific problems for two reasons:

First, security-critical systems run within an adversarial environment, and interact with malicious agents. Malicious agents might perform very specific and possibly intricate actions in order to break the security of such systems.

Second, goals of security-critical systems might prescribe security-specific properties. For instance, confidentiality and authentication goals

are specific to security.

Chapter 4 proposed techniques for detecting vulnerabilities that might cause violations of security properties. Chapter 5 focuses on preventing intentional faults, and is primarily targeted to security goals. Chapter 6 focuses on detecting inconsistencies, which are vulnerabilities.

The boundary between security and other disciplines such as safety might be a bit blurred. A hazardous situation such as a train collision, or a security violation such as a decrease in the balance of an account might both be caused intentionally or accidentally, as summarized in Table 7.1. *Intentional* refers to planned or intended events; *Incidental* refers to events happening by chance [Lap92].

	Hazard	**Violation of security goal**
Accidental fault	train collision due to switch malfunction	cosmic ray or human error causing undue balance decrease
Intentional attack	train collision due to switch sabotage	balance decreased by hacker having penetrated bank system

Table 7.1: Accidental and intentional faults in safety and security-critical systems

Whether a system is security-critical is dependent both on the function of the system and on its environment. Train routing might be seen as a purely safety-critical problem, beside the obvious functional objectives. In the presence of terrorists train routing might require additional measures to ensure that the switches are not configured intentionally to cause a train collision. In this case, is becomes a security-critical system as well.

This thesis does not cover accidental violations of security properties. A good example of accidental fault violating a security goal can be found in [Pou03]. Such faults might be anticipated and prevented through obstacle analysis [vLL00]. Obstacle identification requires some expert knowledge, which is discussed in the next section.

7.2 Linking Modeling Technique to Expert Knowledge

In practice, requirements models are elaborated by experts in security. The purpose of this thesis is to provide engineers with support and techniques for reasoning about models of their requirements. We only focused on model analysis techniques. Expert knowledge and modeling techniques are complementary tools for elaborating requirements.

We have seen in Section 5.6 that experts might benefit from applying some of the techniques proposed here. On the other hand, expert knowledge is required when elaborating the security requirements. This is discussed in this section.

First of all, expert input is explicitly requested in some of the techniques proposed here. For instance, in Section 5.4 our technique requires that the critical security goals of the system-to-be are identified. This requires some expert knowledge about the purpose and nature of the system-to-be.

More generally, any technique that reasons about models requires the model to be accurate to the real world. Incomplete knowledge might lead to requirements models that cannot be implemented with an acceptable level of assurance.

For instance, suppose that an e-purse is required to only sign documents sent within four seconds after it received the correct password. This requirement cannot be implemented with an acceptable level of assurance because the e-purse is to be implemented by a smartcard and the clock driving the CPU embedded into the smartcard is supplied by the smartcard reader in which it is inserted [ISO06]. A potentially malevolent smartcard reader might halt this clock. The result of halting the clock might be that the requirement is violated. Such element of knowledge can only be delivered by experts in security.

Another example has been presented in Section 4.6.3. In this section, we present how additional information about the system-to-be might give rise to additional confidentiality violations. For instance, if the signature of a document can serve to infer the pin code of the signer who signed it, then there is a domain property that must be known to the tool that will search for confidentiality violations. The security expert knows such properties of the domain.

Inaccurate or incomplete information about the real world or about implementation constraints might hinder the ability of elaborating "secure" requirements models. This is why expert knowledge is required during the elaboration of security requirements.

7.3 Open Issues

Specific open issues related with the three contributions of this thesis are discussed in Section 4.8, Section 5.9 and Section 6.8 We present here two general open issues related to security requirements engineering.

7.3.1 Making Formal Specification More Profitable

Formal tools proved to deliver useful support during the requirements engineering phase [PMR+04, HJL96]. They can provide animation, refinement checking, inconsistency detection and anticipation, etc. However, they do not turn out to be much exploited in practice. This notably arises because producing a formal requirements mode is prohibitive. Such formal requirements models only can be analyze by formal tools. Requirements expressed in natural language can also be analyzed, but they need to match some regular grammar that is supported by formal tools [GZ05]. Such grammars are however very similar to the ones defining formal languages.

The problem is that formalizing a requirements model is not profitable enough because formal analysis tool do not share a common language and because virtually no support is available in the redaction of such models.

In order to solve this issue, two complementary approaches can be proposed, and possibly applied together. The first approach is to lower the difficulty of writing formal requirements models. The second approach is to propose standard languages for defining requirements model in order to enable the reuse of models written in this language and automate the tasks to be performed during or after the requirements elaboration phase.

Several techniques have been proposed to make the elaboration of formal models easier. The most recent approach is to generate behavior model from end user scenarios. This approach synthesizes behavior models from simple forms of scenarios [DLD05]. This technique presents scenarios to the user, who is asked to classify them either as positive or negative. A behavior model is incrementally derived from the answers to these scenario-based questions.

Another approach is to provide tool support for automated refinement of goals. Such refinement is driven by a set of refinement patterns [Dar95]. Unfortunately, no tool implements this technique yet.

Another difficulty when elaborating requirements models is to refactor them when a defect is noticed. This refactoring process is rather simple when a state machine captures the requirements model because it generally involves updating some transition of the machine. It can be quite difficult when the requirements model is a goal graph because the refactoring must generally be propagated upwards or downwards in the goal graph. All refinement and goal definitions must be examined, generally with few updated, which could potentially be discharged onto an automated tool.

The other direction for making formal requirements more profitable is to re-use them as much as possible, for instance by generating natural language requirements model from formal requirements model. For instance,

given the formal definition of a goal, the name of the goal and its definition in natural language could be automatically derived, so that the resulting text would be unambiguous. Other documents could be generated, such as user instructions, regulation, etc. Another track is to generate the target application code from such requirements models, by applying synthesis techniques proposed in [PR89a, PR89b, WDR06, BA01].

7.3.2 Deriving Requirements for Security Protocols

This thesis has presented how to derive requirements models for security-critical applications. Such applications generally include security protocols implemented by cryptographic techniques (see Section 2.3.2). The requirements for such protocols can be derived similarly to the ones of agents.

The objective is to choose the adequate protocols for a given application in view of its high-level security goals. It is well-known that a major problems with cryptographic protocol is that they are often inadequately chosen in view of the application-specific security goals [Mea03, And93].

For instance, in the e-purse system, there must be a cryptographic protocol running between an e-purse and a terminal when the e-purse in inserted into the terminal. The purpose of such protocol is notably to enable genuine e-purses and terminals to authenticate to each other. Another requirement of this protocol is to ensure that the terminal is communicating with the e-purse that is inserted into it (see Section 5.6 for this requirement)

A possible track for solving this issue is to represent security protocols as virtual agents that only exist when two real agents communicate with each other. The virtual agent representing the protocol is the intermediary of all communications between the two real agents. For instance, when the e-purse and the terminal authenticate to each other, they both provide the virtual agent with some secret key; the virtual agent compares them and notifies them of the matching of these keys. Of course, the protocol is also responsible for keeping the transmitted keys confidential. The comparison in itself is a leak of information about the keys because a negative answer enable each agent to know that the key transmitted by the opposite agent is not equal to the one transmitted by itself. Additional requirements should be derived in order to solve this issue similarly to the approach of Section 4.6.

This virtual agent can be considered as trustworthy in the eye of every wisher agent that considers any of the two communicating agents trustworthy. The reason is that such virtual agent will actually be refined into two modules, each of which is to be integrated within the communicating agents. Such modules communicate with each other by means of cryptographic protocols. This refinement process belongs to the ad-hoc cryptographic protocol

elaboration and is not part of the requirements elaboration phase.

7.4 Summary of Contributions

7.4.1 Reasoning about confidentiality at requirements engineering time

We propose an extended framework for formally specifying and reasoning about confidentiality requirements in the early phases of software development. Our framework has three components:

- **An axiomatic system** captures how the knowledge of unauthorized agents evolves over time. In a way similar to the "Most Powerful Attacker" hypothesis used in cryptographic protocol analysis, we assume unauthorized agents to be clever and "forgetless" [KMM94, Low96, Bol97, CJM00, Cer03].

  Our axiomatic system (1) lists the cognitive capabilities of such most powerful attackers in the context of requirements engineering; (2) borrows from well-known axiomatic systems for knowledge logics and augments them with axioms binding epistemic logic concepts to concepts from security requirements engineering.

- **A catalog of specification patterns** for confidentiality requirements captures different types of confidentiality requirements on state variables. The patterns are expressed in terms of knowledge operators from our axiomatic system. The catalog has a two-dimensional structure according to the allowed degree of approximate knowledge and its timing. These patterns, in the spirit of [DAC98, CP99], considerably facilitate the task of writing specifications with knowledge operators. They can also be used as a means for eliciting confidentiality requirements and making them more precise.

- **A validation method demonstrated by a proof-of-concept tool** checks requirements models against confidentiality claims expressed in terms of such epistemic patterns. It combines bounded temporal model checking and finite instantiation techniques to translate the confidentiality claims into a constraint satisfaction problem [BCC+03, Jac00]. The latter is solved by constraint programming techniques [Sch02b]. The tool generates counter-example traces leading to violation of a confidentiality claim. In such traces, some agent could get knowledge violating the confidentiality claim. This knowledge is gained according to the axiomatic system describing agents' knowledge capabilities.

The tool also explains the reasoning performed by this agent to infer the forbidden knowledge.

7.4.2 Assigning responsibilities for security requirements

We propose a technique for ensuring that the protection of assets only depends on whether the trustworthy agents enforce the security requirements or assumptions they are responsible for.

Our technique builds on goal-oriented methods. Goals are prescriptive statements of intent about the system-to-be. Security goals prescribe that system assets should be protected against non-authorized use [HLMN06, vL04]. Security goals are enforced through security requirements, assigned to software agents, and security assumptions, assigned to environment agents [vL03b].

Our technique has four components:

- **A goal-oriented model of trustworthiness** identifies trustworthy agents from high-level security goals. Trustworthiness propagates down to software requirements and environment assumptions. Agents that are trustworthy for a security goal will enforce those requirements or assumptions assigned to it which contribute to the security goal.

- **A security-specific criterion for requirements completeness** captures the key idea of our method in terms of our trustworthiness model.

  A set of security requirements and assumptions is *complete* with respect to a set of security goals if the subset of requirements and assumptions assigned to trustworthy agents enforce all the security goals.

- **A set of local constraints on requirements models** is sufficient to guarantee our completeness criterion. These constraints enable local and compositional reasoning about our completeness criterion. Each requirement or assumption that is not assigned to a trustworthy agent is said to be *untrustworthy*. Untrustworthy responsibility assignments must be resolved through the insertion of countermeasures yielding new security requirements. Three classes of countermeasures have been proposed. They borrow from obstacle resolution tactics [vLL00].

- **A constructive and compositional process** for elaborating requirements model is proposed. This process ensures that the delivered requirements model enforces our completeness criterion. This process builds on the proposed local constraints. This process requires that

every responsibility assignment is flagged as trustworthy or not trustworthy. We show that this can be discharged to automated verification tools such as Alloy or PVS [Jac06, SORSC01].

We apply our technique to elicit the security requirements of an e-purse system. Our requirements model is compared to the one prescribed by the European Central Bank (ECB) for such system [ECB03]. This document presents the expectations of the ECB with regard to "*technical security of e-money schemes*". It contains a "*list of security objectives that should be met by e-money schemes in order to cover risks/threats elicited in a comprehensive risk/threat analysis, according to the Common Criteria methodology*" [CC99]. Our approach produces two security requirements that are not mentioned in this document. These requirements are missing because the corresponding threats were not identified. We can produce two attacks on the requirements model based on the fact that these two security requirements are missing.

7.4.3 Detecting divergences in goal models

We propose an automated technique for detecting divergences in goal model. A *divergence* between goals corresponds to situations where some particular combination of circumstances can be found that make the goals logically inconsistent. Such particular combination of circumstances is called *boundary condition* [vLDL98].

Our contribution is as follows:

- **The formal definition of divergence is made further precise.** One of the three formal conditions defining divergences in [vLDL98] states that the boundary condition must be satisfiable in the domain of the system-to-be. We first show that this condition is too weak and therefore that the definition of divergence is also too weak: some goals can be divergence according to the formal definition without being inconsistent. We extend this condition by mentioning that it must be satisfiable *by the agents of the system that will not have any responsibility related to the enforcement of the divergent goals*. We show that this new definition has the soundness property that divergent goals deliver inconsistent requirements.

  This refined condition should be evaluated when the goals are asserted, without having already identified agents and responsibility assignments. We therefore propose to annotate goals with some information that is both easy to obtain at this stage and from which one can approximate the responsibility assingments. This annotation consists in tagging some of the variables referenced by a goal as "intended

to be constrained by the goal". These are said to be the *constrained* variables of the goal [Let01].

Our refined third condition for divergence states that the boundary condition must be *realizable* by agents of the system not involved in the satisfaction of the divergent goals [PR89b, PR89a, PPS06, Let01]. These agents are approximated by the constrained variables.

- **A technique for detecting divergences** involving propositional goals is proposed.

  A brute-force approach for detecting divergence would be to consider that all subsets of goals might be divergent and to test each of these sets in turn for divergence by testing the existence of a realizable boundary condition. Testing the realizability of a boundary condition can be done using standard techniques [PR89b, PR89a, PR90, PPS06, WDR06].

  The realizability-testing problem is known to be very hard. To avoid testing every combination of goals for divergence, we propose a variant of the realizability-testing technique that enable us to detect all divergences included within a given set of goals at once. Our variant requires that the set of goal submitted to it enforces some precondition.

  Based on this adapted divergence-testing technique, we propose an improvement on the brute-force divergence detection. The sets of goals tested for divergence are the maximal sets of goals enforcing the precondition of our divergence-testing technique. These sets are called "*candidate sets*". We proposed three extensions of our technique in order to keep the candidate sets of goals as few and as small as possible.

  The technique has been experimented by means of a prototype tool.

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AEPA01] Annie I. Antón, Julia B. Earp, Colin Potts, and Thomas A. Alspaugh. The role of policy and privacy values in requirements engineering. In *IEEE 5th International Symposium on Requirements Engineering (RE'01)*, pages 138 – 145, Toronto, Canada, August 2001.

[Amo94] E.J. Amoroso. *Fundamentals of Computer Security*. Prentice Hall, 1994.

[AMPS98] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *IFAC Symposium on System Structure and Control*, pages 469 – 474, Elsevier, 1998.

[And93] Ross Anderson. Why cryptosystems fail. In *CCS '93: Proceedings of the 1st ACM conference on Computer and communications security*, pages 215–227, New York, NY, USA, 1993. ACM Press.

[And99] Ross Anderson. How to cheat at the lottery or, massively parallel requirements engineering. invited talk at the 1999 Computer Security Applications Conference, 1999.

[And01] Ross Anderson. *Security Engineering: A guide to building dependable distributes systems*. Wiley, 2001.

[AT04] R. Alur and S. La Torre. Deterministic generators and games for ltl fragments. *ACM Trans. Comput. Log.*, 5(1):1 – 25, 2004.

[BA01] F. Bacchus and M. Ady. Planning with resources and concurrency: A forward chaining approach. In *International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 417 – 424, 2001.

[BAN89] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *Proceedings of the Royal Society of London*, 426:233–271, 1989.

[BC93] P. Bieber and F. Cuppens. Expression of confidentiality policies with deontic logic. *Deontic Logic in Computer Science: Normative System Specification, J.-J.Ch. Meyer and R.J. Wieringa (eds), John Wiley and Sons*, page Ch. 5, 1993.

[BC95] Robert H. Bourdeau and Betty H. C. Cheng. A formal semantics for object model diagrams. *IEEE Trans. Softw. Eng.*, 21(10):799–821, 1995.

[BCC+99] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 317 – 320, New Orleans, Louisiana, United States, 1999. ACM Press, New York, NY, USA.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[BDD+98] P. Bertrand, R. Darimont, E. Delor, P. Massonet, and A. van Lamsweerde. Grail/kaos: an environment for goal drivent requirements engineering. In *Proceedings ICSE'98 - 20th International Conference on Software Engineering*, Kyoto, April 1998. IEEE-ACM.

[BDL06] David Basin, Jürgen Doser, and Torsten Lodderstedt. Model driven security: From uml models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, 2006.

[BDLN05] Haley Charles B., Moffett Jonathan D., Robin Laney, and Bashar Nuseibeh. Arguing security: validating security requirements using structured argumentation. In *Proc. Third Symposium on Requirements Engineering for Information Security (SREIS'05) held in conjunction with the 13th International Requirements Engineering Conference (RE'05)*, Paris, France, 2005.

[BDR04] G. Barthe, P.R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop,* June 28 - 30, 2004, Pacific Grove, California, USA, pages 100–114. IEEE Computer Society Press, 2004.

[BG89] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2), 1989.

[BGH+00] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, E. Van Herreveghen, and M. Waidner. ikp - a family of secure electronic payment protocols. *IEEE Journal on Selected Areas in Communications*, 18(4):611 – 627, 2000.

[Bib75] Ken Biba. Integrity consideration for secure computer systems. Technical Report MTR31-53, Mitre Corporation, 1975.

[Bis04] Matt Bishop. *Computer Security*. Addison-Wesley, 2004.

[BL73] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corperation, Bedford, MA, 1973.

[BN89] Dr. David F.C. Brewer and Dr. Micheal J. Nash. The chinese wall security policy. *sp*, 00:206, 1989.

[Boa00] Maftia Project Board. Maftia deliverable d1: Reference model and use cases. Technical report, University of Newcastle upon Tyne (UK), FCUL, Lisboa (P), DERA, Malvern (UK), Universität des Saarlandes (D), CNRS-LAAS (F), IBM Zurich Research Lab (CH), 2000. http://www.maftia.org/deliverables/D1.pdf.

[Bol97] D. Bolignano. Towards a mechanization of cryptographic protocol verification. In *Proc. 9th International Computer Aided Verification Conference*, pages 131–142, 1997.

[Bor95] Beizer Boris. *Black-box testing - Techniques for functional testing of software and syatems*. John Wiley & Sons, 1995.

[BPG+04] Paolo Bresciani, Anna Perini, Paolo Giorgini, Fausto Giunchiglia, and John Mylopoulos. Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677 – 691, 1986.

[cam] Cambridge dictionaries online. http://dictionary.cambridge.org/.

[Cap04] L. Capra. Engineering human trust in mobile system collaborations. In *In Proc. of the 12th International Symposium on the Foundations of Software Engineering (SIGSOFT 2004/FSE-12)*, Newport Beach, CA, November 2004.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CC99] Common criteria for information technology security evaluation, version 2.1. http://www.commoncriteria.org/, 1999.

[CCC+04] Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, J. Mantovani, S. Mödersheim, and L. Vigneron. A high level protocol specification language for industrial security-sensitive protocols. In Australian Computer Society, editor, *Proc SAPS'04*, 2004.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *In Logic of Programs, Workshop*, 1982.

[CER] Cert statistics. http://www.cert.org/stats/cert_stats.html.

[Cer03] Iliano Cervesato. Data access specification and the most powerful symbolic attacker in msr. In *Software Security - Theories and Systems — ISSS 2002*, pages 384–416, Tokyo, Japan, November 2003. Springer-Verlag LNCS 2609.

[CFL+97] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. Referee: Trust management for web applications. *The World Wide Web Journal*, 2(3):127–139, 1997.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. the MIT Press, January 2000. ISBN 0-262-03270-8.

[CH02] Yunja Choi and Mats P.E. Heimdahl. Model checking rsml-e requirements. *hase*, 00:109, 2002.

[CILN02] Robert Crook, Darrel C. Ince, Luncheng Lin, and Bashar Nuseibeh. Security requirements engineering: When anti-requirements hit the fan. In *RE '02: Proceedings of the 10th*

Anniversary IEEE Joint International Conference on Requirements Engineering, pages 203–205, Washington, DC, USA, 2002. IEEE Computer Society.

[CJM00] E.M. Clarke, S. Jha, and W. Marrero. Verifying security protocols with brutus. *ACM Transactions on Software Engineering and Methodology(TOSEM)*, pages 443–487, October 2000.

[CL95] F. Cristian and M.R. Lyu. *Software Fault Tolerance*, chapter Exception Handling. Wiley, 1995.

[CNYM00] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering.* Kluwer Academic Publishers, Boston, 2000.

[CP99] Marsha Chechik and Dimitrie O. Paun. Events in property patterns. In *SPIN*, pages 154–167, 1999.

[CPP06] Shari Lawrence Pfleeger Charles P. Pfleeger. *Security in Computing, 4th Edition.* Prentice Hall, 2006.

[CRS04] Alessandro Cimatti, Marco Roveri, and Daniel Sheridan. Bounded verification of past ltl. In *Proc. of Formal Methods in Computer-Aided Design: 5th International Conference, FMCAD 2004*, Austin, Texas, USA, November 2004.

[DAC98] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *Proceedings of the second workshop on Formal methods in software practice*, pages 7–15. ACM Press, 1998.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE).* ACM Press, 2001.

[Dar95] R. Darimont. *Process Support for Requirements Elaboration.* PhD thesis, Université catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, 1995.

[DD77] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.

[DDD05] G. Dooms, Y. Deville, and P. Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In *11th International Conference on Principles and Practice of Constraint Programming*, 2005.

[DDL+02] N. Damianou, N. Dulay, E. Lupu, M. Sloman, and T. Tonouchi. Tools for domain-based policy management of distributed system. In *Proc. IEEE/IFIP Network Operations and Management Symposium (NOMS2002)*, Florence, Italy, 2002.

[DDLM00] N. Damianou, N. Dulay, E. Lupu, and M.Sloman. Ponder: A language for specifying security and management policies for distributed systems, v 2.3. Technical Report DoC 2000/1, Imperial College, 2000.

[Dem04] Robert Demolombe. Reasoning about trust: A formal logical framework. In *Trust Management: Second International Conference, iTrust 2004*, volume 2995, pages 291–303. Springer Lecture Notes in Computer Science, 2004.

[Den87] D Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, 13(2):222 – 232, 1987.

[DFvL91] A. Dardenne, S. Fickas, and A. van Lamsweerde. Goal-directed concept acquisition in requirements elicitation. In *Proc. IWSSD-6 - 6th Intl. Workshop on Software Specification and Design*, pages 14–21, Como, 1991.

[dlBSW03] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 32 – 43, Uppsala, Sweden, 2003. ACM Press.

[DLD05] Christophe Damas, Bernard Lambeau, and Pierre Dupont. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng.*, 31(12):1056–1073, 2005. Member-Axel van Lamsweerde.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael*. Springer, 2002. ISBN 3-540-42580-2.

[DS00] Premkumar T. Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 227–239, New York, NY, USA, 2000. ACM Press.

[DvL96] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. In *Proc. FSE4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, pages 179–190, San Francisco, October 1996.

[DvLF93] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20:3 – 50, 1993.

[DW87] Clark D. and D. Wilson. A comparison of commercial and military security policies. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, California, 1987.

[DY81] D. Dolev and A.C. Yao. On the security of public key protocols. In *Proceedings of the IEEE 22nd Annual Symposium on Foundations of Computer Science*, pages 350 – 357, 1981.

[ECB03] European Central Bank ECB. Electronic money system security objectives according to the common criteria methodology, 2003.

[ECR06] ECRYPT. Yearly report on algorithms and key sizes. Technical report, D.SPA.16 Rev 1.0, IST-2002-507932, 2006.

[EN] John Ellson and Stephen North. Graphviz - graph visualization software. http://www.graphviz.org/.

[Eng98] Joeri Engelfriet. Monotonicity and persistence in preferential logics. *J. Artif. Intell. Res. (JAIR)*, 8:1–21, 1998.

[FDP01] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic*, 2(1), January 2001.

[Fea87] M.S. Feather. Language support for the specification and development of composite systems. *ACM Transactions on Programming Languages and Systems*, 9(2):198 – 234, april 1987.

[FHMV95] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[Fir03] Donald G. Firesmith. Analyzing and specifying reusable security requirements. In *proc. of International Workshop on Requirements for High Assurance Systems*, Monterey, California, September 9 2003.

[Fir04] Donald Firesmith. Specifying reusable security requirements. *Journal of Object Technology*, 3(1):61 – 75, 2004.

[Gar99a] F.C. Gartner. Fundamentals of fault-tolerant distributed computing in asynchronous environment. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[Gar99b] Stoneburner Gary. Cspp- guidance for cots security protection profiles. Technical Report NISTIR 6462, National Institutes of Standards and Technology (NIST), U.S. Department of Commerce, 1999.

[GMMZ04a] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Filling the gap between requirements engineering and public key/trust management infrastructures. In *Proc. of EuroPKI'04, LNCS 3093*, pages 98 – 111. Springer-Verlag, 2004.

[GMMZ04b] Paolo Giorgini, Fabio Massacci, John Mylopoulos, and Nicola Zannone. Requirements engineering meets trust management: Model, methodology, and reasoning. In *Proc. of iTrust'04, LNCS 2995*, pages 176 – 190. Springer-Verlag, 2004.

[GMMZ05a] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Modeling security requirements through ownership, permission and delegation. In *Proc. RE'05*, pages 167 – 176, Paris, 2005.

[GMMZ05b] Paolo Giorgini, Fabio Massacci, John Mylopoulos, and Nicola Zannone. St-tool: A case tool for security requirements engineering. In *Proc. RE'05*, pages 451 – 452, 2005.

[GMMZ06] Paolo Giorgini, Fabio Massacci, John Mylopoulos, and Nicola Zannone. Detecting conflicts of interest. In *Proc. RE'06*, pages 308 – 311, 2006.

[GPSAK06] Suhas Gupta, Pranay Pratap, Huzur Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 65–72, New York, NY, USA, 2006. ACM Press.

[GQ94] L. Gong and X. Qian. The complexity and composability of secure interoperation. In *Proc. IEEE Symposium on Security and Privacy*, pages 190 – 200, 1994.

[GQ96] Li Gong and Xiaolei Qian. Computational issues in secure interoperation. *IEEE Transactions on Software Engineering*, 22(1):43 – 52, 1996.

[Gro04] The Open Group. *Security Design Patterns*. 2004. http://www.opengroup.org/publications/catalog/g031.htm.

[GS02] Tyrone Grandison and Morris Sloman. Specifying and analysing trust for internet applications. In *Proceedings of*

the second IFIP Conference on E-Commerce, E-Business, E-Government (I3E 2002), pages 145–157, Lisbon, Portugal, October 7-9 2002.

[GZ05] Vincenzo Gervasi and Didar Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM Transaction Software Engineering Methodologies*, 14(3), 2005.

[HJL96] C. Heitmeyer, R.D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Trans.on Software Eng. and Methodology*, 5(3):231–261, July 1996.

[HLMN06] B. Haley, C. Laney, D. Moffett, and Bashar Nuseibeh. Using trust assumptions with security requirements. *Requir. Eng.*, 11(2):138–151, 2006.

[HM85] J. Halpern and Y. Moses. *Towards a theory of knowledge and ignorance: preliminary report*, pages 459–476. Springer-Verlag New York, Inc., 1985.

[HMLN06] Charles B. Haley, Jonathan D. Moffett, Robin Laney, and Bashar Nuseibeh. A framework for security requirements engineering. In *Proc. Software Engineering for Secure Systems Workshop (SESS'06), co-located with the 28th International Conference on Software Engineering (ICSE'06)*, Shanghai, China, 2006.

[HR00] Klaus Havelund and Grigore Roşu. Testing linear temporal logic formulae on finite execution traces. Technical Report TR 01-08, RIACS, May 2001. Written 20 December 2000.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[ISO06] Iso7816 identification cards – integrated circuit cards – part 3: Cards with contacts – electrical interface and transmission protocols, 2006. Published standard.

[Jac75] M. Jackson. *Principles of Program Design*. Academic Press, 1975.

[Jac89] Jeremy Jacob. On the derivation of secure components. In *Proc. of 1989 IEEE Symposium on Security and Privacy*, page 242, Oakland, CA, may 1989.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, November 2000.

[Jac01] Michael Jackson. *Problem Frames*. Addison Wesley, 2001.

[Jac06] Daniel Jackson. *Software Abstractions Logic, Language, and Analysis*. MIT Press, 2006.

[JAL04] Mark Jago, Natasha Alechina, and Brian Logan. A complete and decidable logic for resource bounded agents. In *proc. AAMAS 04*, pages 606 – 613, New York, July 2004. Columbia Univseristy.

[JCN92] J.Mylopoulos, L. Chung, and B. Nixon. Representing and using non-functional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering, Special Issue on Knowledge Representation and Reasoning in Software Development*, pages 483–497, June 1992.

[Jos96] M. Joseph. *Real-time Systems Specification, Verification and Analysis*. Prentice Hall Intl, 1996.

[Jun01] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, aug 2001.

[JvdMRV04] Halpern J., van der Meyden R., and Vardi. Complete axiomatizations for reasoning about knowledge and time. *SIAM Journal on Computing*, 33(3):674 – 703, 2004.

[JZ93] M. Jackson and P. Zave. Domain descriptions. In *Proc. first IEEE International Symposium on Requirements Engineering (RE93)*, pages 56 – 64. IEEE Computer Society Press, 1993.

[Kem03] Richard A. Kemmerer. Cybersecurity. In *Proceedings of the 25th international conference on Software engineering*, pages 705 – 715, Portland, Oregon, 2003. IEEE.

[Ker83] Auguste Kerkhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, 9th serie:(Jan) pp 5–38 and (Feb) pp 161–191, 1883. http://www.cl.cam.ac.uk/ fapp2/kerckhoffs/.

[KMM94] R. Kemmerer, C. Meadows, and J. Millen. Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, 1994.

[Koy92] R. Koymans. *Specifying message passing and time-critical systems with temporal logic*. Number LNCS 651. Springer-Verlag, 1992.

[Lan04] Renaud De Landtsheer. Solving csp including a universal quantification. In *Proc. of the Second International Mozart/Oz conference*, 2004.

[Lap92] J.C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

[Len04] Arjen K. Lenstra. *Handbook of Information Security*. Wiley, 2004.

[Let01] E. Letier. *Reasoning about Agents in Goal-Oriented Requirements Engineering*. PhD thesis, Université Catholique de Louvain, Dépt. Ingénierie Informatique, Louvain-la-Neuve, Belgium, May 2001.

[Lev95] N.G. Leveson. *Safeware - System Safety and Computers*. Addison-Wesley, 1995.

[Lip82] Steven B. Lipner. Non-discretionary controls for commercial applications. In *IEEE Symposium on Security and Privacy*, pages 2–10, 1982.

[Liu07] Hao Liu. Développement d'un outil de vérification de cahier des charges pour applications critiques en sécurité. Master's thesis, Universtité catholique de Louvain, Département d'ingéniérie Informatique, 2007.

[LKMU05] E. Letier, Jeff Kramer, Jeff Magee, and Sebastian Uchitel. Fluent temporal logic for discrete-time event-based models. In *Proceedings ESEC/FSE 2005 - 5th joint meeting of the the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon, Portugal, 2005.

[LNIJ04] Luncheng Lin, Bashar Nuseibeh, Darrel Ince, and Michael Jackson. Using abuse frames to bound the scope of security problems. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International (RE'04)*, pages 354–355, Washington, DC, USA, 2004. IEEE Computer Society.

[Lon] David Long. The bdd library. http://www.cs.cmu.edu/ modelcheck/bdd.html.

[Low96] G. Lowe. Breaking and fixing the needham-schroeder publickey protocol using fdr. In *Tools and Algorithms for the Construction and Analysis of Systems*, 1996.

[LS99] E. Lupu and M. Sloman. Conflicts in policy-based distributed systems management. *IEEE Trans. on Software Engineering*, 25(6):852 – 869, 1999.

[LS04] Bev Littlewood and Lorenzo Strigini. Redundancy and diversity in security. In *Proc. 9th European Symposium on Research in Computer Security*, pages 423 – 438, Sophia Antipolis, 2004.

[LV00] Arjen K. Lenstra and Eric R. Verheul. Selecting cryptographic key sizes. In *PKC2000*, pages 446 – 465, 2000.

[LvL02] E. Letier and A. van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings ICSE'2002 - 24th International Conference on Software Engineering*, May 2002.

[LvL04] Emmanuel Letier and Axel van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 53–62. ACM Press, 2004.

[LvL05] Renaud De Landtsheer and Axel van Lamsweerde. Reasoning about confidentiality at requirements engineering time. In *Proceedings ESEC/FSE'05 Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lisbon Portugal, September 2005.

[LYM03] Lin Liu, Eric Yu, and John Mylopoulos. Security and privacy requirements analysis within a social setting. In *Proc. 11th IEEE International Requirements Engineering Conference (RE'03)*, page 151, Monterey Bay, California, USA, September 2003. IEEE Computer Society Press.

[Man00] Daniel Manchala. E-commerce trust metrics and models. *IEEE Internet Computing*, March-April 2000.

[Mau96] Ueli M. Maurer. Modelling a public-key infrastructure. In *Proceedings of the 4th European Symposium on Research in Computer Security*, pages 325–350. Springer-Verlag, 1996.

[Mea03] Catherine Meadows. Some issues in the specification of requirements for cryptographic protocols: a position paper. In *Proceedings of the RE'03 Workshop on Requirements for High Assurance Systems (RHAS'03)*, pages 31 – 32, Monterey (CA), September 2003. IEEE-ACM.

[MEL01] AP. Moore, R.J. Ellison, and R.C. Linger. Attack modeling for information security and survivability. Technical Report CMU/SEI-2001-TN-001, CMU, 2001.

[Mey07] Julien Meyer. Développement d'un outil de vérification de cahier des charges pour applications critiques en sécurité. Master's thesis, Universtité catholique de Louvain, Département d'ingéniérie Informatique, 2007.

[MF99] John McDermott and Chris Fox. Using abuse case models for security requirements analysis. In *ACSAC '99: Proceedings of the 15th Annual Computer Security Applications Conference*, page 55, Washington, DC, USA, 1999. IEEE Computer Society.

[MHI05] Nancy R. Mead, Eric D. Hough, and Theodore R. Stehney II. Security quality requirements engineering. Technical Report CMU/SEI-2005-TR-009, CMU, 2005.

[MK99] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. Wiley, 1999.

[MO05] Sjouke Mauw and Martijn Oostdijk. Foundations of attack trees. In *ICISC*, pages 186–198, 2005.

[Mor94] Peter Morris. *Introduction to Game Theory*. Springer-Verlag, 1994.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[MW84] Zohar Manna and Pierre Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 6(1):68 – 93, 1984.

[MZ06] F. Massacci and N. Zannone. Detecting conflicts between functional and security requirements with secure tropos: John rusnak and the allied irish bank. Technical Report DIT-06-002, University of Trento, 2006.

[nin] Survey of papers from the itrust 2003 and 2004 trust management conferences. http://www.ninebynine.org/iTrust/iTrust-survey.html.

[NIS06] NIST. Recommendation for key management. *NIST Special Publication 800-57*, 2006.

[NSA05] NSA. Fact sheet suite b cryptography. In *Proc. RSA Conference*, 2005.

[obj] The objectiver re tool. www.objectiver.com.

[ODK99] Rodolphe Ortalo, Yves Deswarte, and Mohamed Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Trans. Softw. Eng.*, 25(5):633–650, 1999.

[Ort98] Rodolphe Ortalo. A flexible method for information system security policy specification. In *Proc. of the 5th European Symposium on Research in Computer Security*, pages 67 – 84. Springer-Verlag, 1998.

[oSET00] ACM Transactions on Software Engineering and Methodology (TOSEM), editors. *Special Issue on Software Engineering and Security*. ACM, October 2000.

[oST] National Institute of Standard and Technology. Data encryption standard (des). http://csrc.nist.gov/encryption. FIPS 46; DRAFT FIPS46-3, incorporating upgrade to 3DES.

[Pap94] Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[PMR+04] Ch. Ponsard, P. Massonet, A. Rifaut, J.F. Molderez, A. van Lamsweerde, and H. Tran Van. Early verification and validation of mission-critical systems. In *Proceedings of FMICS'04, 9th International Workshop on Formal Methods for Industrial Critical Systems*, Linz (Austria), 2004.

[Pnu98] A. Pnueli. Verification by finitary abstraction. In *Proc. SPIN'98: 4th Intl. SPIN Workshop*, Paris, 1998.

[Pou03] Poureva.org. Le ministre dewael reconnait la faillibilité du vote électronique grâce à un rayon cosmique complice !, 2003. (in french) http://www.poureva.org/article.php3?id_article=36.

[PPS06] Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. In *Seventh International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 06)*, Charleston, South Carolina, USA, 2006.

[PR89a] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of the 16th ACM Symposium on Principles of Programming Languages*, pages 179 – 190, 1989.

[PR89b] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In *ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming*, pages 652 – 671, 1989.

[PR90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proc. of 31st IEEE Symp. on Foundation of Computer Science*, pages 746 – 757, 1990.

[pro] The proton electronic purse system. http://protonworld.st.com/. see also http://www.proton.be/.

[PS05] F. M. Puchkov and K. A. Shapchenko. Static analysis method for detecting buffer overflow vulnerabilities. *Program. Comput. Softw.*, 31(4):179–189, 2005.

[PST91] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall, 1991.

[REA04] De Landtsheer Renaud, Letier Emmanuel, and Van Lamsweerde Axel. Deriving tabular event-based specifictions from goal-oriented requirements models. *Requirements Engineering Journal*, pages 104 – 120, 2004.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, march 2004. ISBN: 0-262-22069-5.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (second edition).* Prentice Hall, 2003.

[Rob89] W.N. Robinson. Integrating multiple specifications using domain goals. In *Proc. IWSSD-5 - 5th Intl. Workshop on Software Specification and Design, IEEE*, pages 219 – 225, 1989.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adelman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 2(21):120–126, Feb 1978.

[Sch96] B. Schneier. *Applied Cryptography.* Wiley, 1996.

[Sch97] B. Schneier. Why criptography is harder than it looks. Technical report, COUNTERPANE SYSTEMS, 1997. http://www.counterpane.com /whycrypto.html.

[Sch00] Bruce Schneier. *Secret and Lies.* Wiley, 2000.

[Sch02a] Pierre-Yves Schobbens. Realizing kaos agents. In *Proc. Symposium on Logic in Games and Multiagent Systems*, 2002.

[Sch02b] Christian Schulte. *Programming Constraint Services*, volume 2302 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, Berlin, Germany, 2002.

[scr06] Formal specification and verification of data separation in a separation kernel for an embedded system. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS 2006)*, pages 346 – 355, Alexandria, VA, 2006.

[SFO03] Guttorm Sindre, Donald G. Firesmith, and Andreas L. Opdahl. A reuse-based approach to determining security requirements. In *Proc. 9th International Workshop on Requirements Engineering: Foundation For Software Quality (REQFS'03)*, Austria, 2003.

[SG01] Zisman A. Spanoudakis G. *Inconsistency Management in Software Engineering: Survey and Open Research Issues*, pages 329 – 380. World Scientific Pub. Co., 2001. ISBN 981-02-4973.

[Sin06] Guttorm Sindre. Trust-related requirements: a taxonomy. In IKnapp & Magyar, editor, *Proc 15th Int'l Conf on Information Systems Development (ISD'06)*, Budapest, Hungary, 2006. Springer.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. Boston: Thomson Course Technology, 2006.

[SJW02] Oleg Sheyner, Somesh Jha, and Jeannette M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[SO05] Guttorm Sindre and Andreas L. Opdahl. Eliciting security requirements with misuse cases. *Requir. Eng.*, 10(1):34–44, 2005.

[SORSC01] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Prover Guide*. SRI International, Computer Science Laboratory, 333 Ravenswood Avenue, Menlo Park CA 94025, November 2001. http://pvs.csl.sri.com/doc/pvs-prover-guide.pdf.

[Spi02] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Professional, 2002.

[SR05] Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in capability-based systems. In *TGC*, pages 248–278, 2005.

[SS74] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Communications of the ACM*, 17(7), July 1974.

[TL04] N. Tuptuk and E. Lupu(ed.). Trustcom project - state of the art evaluation. Technical report, Imperial College, London, June 2004.

[VBBC04] Nicolas Vanderavero, Xavier Brouckaert, Olivier Bonaventure, and Baudouin Le Charlier. The honeytank: a scalable approach to collect malicious internet traffic. In *International Infrastructure Survivability Workshop (IISW'04)*, Lisbon, Portugal, December 2004.

[VGV92] J. Verschuren, R. Govaerts, and J. Vandewalle. Simultaneous enforcement of the bell-lapadula and the biba securitypolicy models in an osi-distributed system. In *Singapore ICCS/ISITA '92. 'Communications on the Move'*, 1992.

[vL00] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Invited Paper for ICSE'2000 - 22nd International Conference on Software Engineering,ACM Press*, Limerick, 2000.

[vL01] Axel van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE'01 - 5th IEEE International Symposium on Requirements Engineering*, pages 249–263, Toronto, August 2001.

[vL03a] A. van Lamsweerde. The kaos meta-model ten years after. Technical report, Universite catholique de Louvain - Departement d'ingéniérie informatique, april 2003.

[vL03b] Axel van Lamsweerde. From system goals to software architecture. In *SFM*, pages 25–43, 2003.

[vL04] Axel van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *Proceedings of the 26th International Conference on Software Engineering*, pages 148–157. IEEE Computer Society, 2004.

[vLDL98] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development*, pages 908 – 926, November 1998.

[vLDM95] A. van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: Problems and lessons learned. In *Proc. RE'95 - 2nd Int. Symp. on Requirements Engineering*, York, 1995. IEEE.

[vLL00] A. van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering, Special Issue on Exception Handling*, 26(10):978 – 1005, October 2000.

[VM01] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, 2001.

[WDR06] Martin De Wulf, Laurent Doyen, and Jean-François Raskin. A lattice theory for solving games of imperfect information. In *Proceedings of HSCC 2006, LNCS 3927*, Santa Barbara, 2006.

[WG05] J. Wilander and J. Gustavsson. Security requirements—a field study of current practice. In *E-Proceedings of the Symposium on Requirements Engineering for Information Security, in conjunction with the 13th IEEE International Requirements Engineering Conference*, Paris, 2005.

[Win98] J. Wing. A symbiotic relationship between formal methods and security. In *Proc. NSF Workshop on Computer Security, Fault Tolerance, and Software Assurance: From Needs to Solution*, December 1998.

[Win03] Jeannette M. Wing. Beyond the horizon: A call to arms. *IEEE Security and Privacy*, pages 62 – 67, November/December 2003.

[Wor97] DIMACS Workshop. *Proceedings of the DIMACS Workshop on Design and Formal Verification of Security Protocols.* Rutgers University, September 1997.

[WPJH98] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. Scenario usage in system development: A report on current practice. In *ICRE '98: Proceedings of the 3rd International Conference on Requirements Engineering*, page 222, Washington, DC, USA, 1998. IEEE Computer Society.

[YL00] Eric S. K. Yu and Lin Liu. Modelling trust for system design using the i * strategic actors framework. In *Trust in Cybersocieties*, pages 175–194, 2000.

[Yu95] Eric Yu. *Modelling Strategic Relationships for Process Reengineering*. PhD thesis, Dept. of Computer Science, University of Toronto, 1995.

[Yu97] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97)*, pages 226–235, Washington D.C., USA, January 1997. IEEE Computer Society Press.

[Yue87] K. Yue. What does it mean to say that a specification is complete? In *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, 1987.

[ZJ97] P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Software Eng. and Methodology*, 6(1):1–30, Jan 1997.

List of Figures

1.1 Statistics of Reported Vulnerabilities [CER] 9
1.2 Software layers size compared to their security assurance . . . 9

2.1 Schematic representation of the BLP security policy model . 26

3.1 The meta, domain and instance levels 35
3.2 Structuring the KAOS meta-model 36
3.3 Diagrammatic representation of requirements models 37
3.4 Goal graph refining a security goal of the e-signature system . 43
3.5 Goal graph refining a functional goal of the e-signature system 44
3.6 Divergence in the e-Signature System 45
3.7 Goal-oriented elaboration of requirements method 50
3.8 Refining the goal *Achieve[Wanted Item Paid]* 53
3.9 Divergent goals from the e-purse system 54
3.10 Alternative refinement for goal *Achieve[Wanted Item Paid]* . 54
3.11 Responsibility assignment in the e-purse system 55
3.12 Obstacle analysis on an assumption 55
3.13 Strategies for resolving obstacles 56
3.14 Resolving the obstacle *Chew Blocked In Terminal* 56

4.1 Axioms of the S5 epistemic logic system 62
4.2 Axioms for temporal knowledge 62
4.3 Object model fragment for the e-purse payment system . . . 71
4.4 Counter-example trace produced by CONCHITA 72
4.5 Graphical view of the output trace 72
4.6 Explaining the reasoning for disclosure 73
4.7 CONCHITA's Overall Architecture 75
4.8 Assumption Table of State Variable x 84
4.9 Translating assumption tables to non-temporal assumptions . 84
4.10 Translating claim tables to non-temporal claims 86
4.11 Confidentiality requirement enforcement process 97
4.12 Object model of the e-signature system 99

5.1 Trustworthiness diagram 112
5.2 Distrust condition 114

5.3 Srong mitigation of untrustworthy assignement 118
5.4 Weak mitigation of untrustworthy assignement 119
5.5 Untrustworthiness prevention in assignment 120
5.6 Elaborating a secure requirements model 121
5.7 Discharging the test of trustworthiness to an automated tool 125
5.8 Explanation of untrustworthiness given by Alloy 127
5.9 Critical Security Goals of Electronic Payment Systems 135
5.10 Refining the buyer's critical security goal 135
5.11 Ensuring an adequate payment amount 137
5.12 Refining the terminal authentication countermeasure 138

6.1 Validating the amount encoded in an e-Purse terminal 143
6.2 Divergence in the e-Signature System 144
6.3 Context diagram of a mine pump System 145
6.4 Goal graph fragment for the mine pump system 146
6.5 Transition system for the *PumpController* agent 147
6.6 Four divergences from the mine pump system 154
6.7 Structuring the divergences of the mine pump system 155
6.8 A boundary trace for the mine pump system divergence . . . 165
6.9 An unnecessarily deep boundary trace 167
6.10 A shallow and narrow boundary trace 167
6.11 Opposition Graph of the Mine Pump system 169
6.12 Sample Opposition Graphs 170
6.13 Divergences delivered by SSV on the MinePump 174
6.14 Non-simplified boundary trace for the MinePump 175
6.15 Graphical output of SSV for the MinePump 175
6.16 First snapshot of train control system 177
6.17 Boundary trace of the first snapshot 177
6.18 Second snapshot of train control system 178
6.19 Third snapshot of train control system 178
6.20 Boundary trace of the third snapshot 179
6.21 Fourth snapshot of train control system 179
6.22 Fifth snapshot of train control system 180
6.23 Sixth snapshot of train control system 181
6.24 Seventh snapshot of train control system 182
6.25 Eight snapshot of train control system 183
6.26 Ninth snapshot of train control system 184
6.27 Tenth snapshot of train control system 185
6.28 Eleventh snapshot of train control system 186
6.29 Twelfth snapshot of the train control system. 187
6.30 Run time of divergence detection on the Train control system 189
6.31 Measuring the speedup provided by enhanced pruning 189

A.1 Temporal operators supported by CONCHITA 226

A.2 Logical operators supported by CONCHITA 226
A.3 Confidentiality patterns supported by CONCHITA 228
A.4 Temporal exponent supported by CONCHITA 229

Index

abstract signature, 129
abuse case, 24
abuse frames, 24
accessibility relation, 60
Accountability, 20
Adequacy, 21
agent interface model, 37
agent responsibility model, 37
Agents, 38
Alloy, 126
AND-refinement, 43
Asset Owner, 108
Associations, 39
assumption, 45
asynchronous agents, 48
attack, 23
attack graph, 24
Attack patterns, 24
attack trees, 24
Autenticity, 20
Authentication, 21
authorized agent, 10
availability, 10

B method, 150
BAN logics, 18
Bell-la-padula, 25
Biba, 25
blocking state, 162
boundary condition, 44, 151, 156
boundary condition avoidance, 193

C.I.A., 10
candidate sets, 159
Chinese Wall, 25
Clarck-Wilson, 25
common criteria methodology, 23
complete, 108
Complete mediation, 22
Completeness, 21
completeness criterion, 31
CONCHITA , 69
confidentiality, 10
confidentiality checking, 69
conflict, 44
Conflict set, 96
Consistency, 21
constrained negatively, 169
constrained positively, 169
constrained variable opposition graph, 169
constrained variables, 149
Contribution, 115
Control, 46
controllable predecessor, 162
Countermeasure, 115
critical security Goal, 111
Critical security goals, 108

definition, 38, 40
Dependability, 20
Direct Contribution, 115
Disclosure, 10
distribution axiom, 62
Distrust condition, 113
divergence, 44, 146, 151
divergent states, 162
domain model, 34
domain properties, 41
domain-dependent realizability, 48
domain-independent realizability, 48

dominate, 115
Dominate Contribution, 115

Economy of mechanisms, 22
entity, 38
equisatisfiable, 161
eventuality clause, 161
External confidentiality, 23

Fail-safe defaults, 22
formal def, 40
Functional requirements, 10

goal, 40
goal model, 36

Honeypot, 19

i*, 33
I* framework, 27
Implied divergence, 154
Incidental, 195
Incompleteness, 11
Inconsistencies, 11
induced subgraph, 169
instance model, 34
instantaneous confidentiality, 68
integrity, 10
Intentional, 195
inter-model consistency rule, 37
inter-model consistency rules, 36
inter-model rules, 37
Internal confidentiality, 23
Intrusion detection, 19

Kerkhoffs' principle, 18, 63
Knowledge generalization axiom, 62
Kripke structure, 60

Leaf divergence, 154
Least common mechanisms, 22
Least privilege, 22
Lipner, 25

members, 39
memory variables, 161
meta-constraints, 34
meta-model, 34
Minimal conflict set, 96
mitigation, 117, 118
Monitoring, 46
Most knowledgeable asset owner model, 109
Most Powerful Attacker Model, 18
Most Powerful Attacker model, 62

name, 38, 40
Negative introspection axiom, 62
NFR, 33
no learning axiom, 62
Non-critical security goals, 108
Non-repudiation, 20

object, 37
object model, 36
obstacle, 45
Obstacles, 45
obstacles, 193
Open design, 22
open design, 28
operation model, 50

pattern, 40
Perfect recall axiom, 62
Perfect System Knowledge, 63
Physical security, 17
Ponder, 25
Positive introspection axiom, 62
possible worlds, 60
Precision, 21
Privacy, 20
Protection profiles, 25
Psychological acceptability, 23

QBF solvers, 93

realizability, 48
reference goal, 114
relative trustworthiness, 108
requirement, 45

Requirements, 10
Responsibility, 45
role, 39
Root divergence, 154

S5 axiomatic system, 61
secure, 112
Secure Tropos framework, 27
Securely complete, 112
Securely enforced, 112
Securely enforced from viewpoint, 112
security failure, 21
Security goals, 41
security goals, 20
security policy, 25
Security requirements, 10
security target, 20
Separate Normal Form, 161
Separation of privilege, 22
SNF, 161
Socio-technical and organizational security, 17
Software security, 17
spurious divergences, 153
SQUARE, 29
SSV , 172
start clause, 161
states, 60
step clause, 161
strong mitigation, 117
strong realizability, 48
synchronous agents, 48
synchrony assumption, 48

threat, 23
threats, 23
trace, 44
Tropos, 33
Tropos framework, 27
Trust, 20
trust, 27
Trustworthy, 111
trustworthy, 27
Truth axiom, 62

unique control, 47
untrustworthiness prevention, 119
Untrustworthy Assignment, 113
untrustworthy assignment, 109

vocabulary, 149
vulnerability, 21

weak mitigation, 118
weak realizability, 48
weakest boundary condition, 155
Wish, 110

Appendix A

Annex of Chapter 4

A.1 BNF Grammar of CONCHITA

The BNF grammar used to parse the input file of the model checker prototype is described below. We give a simplified version here. The implemented grammar is slightly different because some grammatical elements have been added for algorithmic reasons.

We use the convention that elements in single quotes are syntactic elements, elements in double quotes are identifiers (but we give them a more explicit name than *identifier* for clarity) and other elements are defined by a grammar rule. We use the regular expression notation for choice (`[choice1 | choice2]`), repetition (`*`) and repetition with at least one element (+).

We also use the special keywords `integer`. The keyword `infinity` is tokenized as a special integer representing ∞.

Comments can be inserted in the input file. The character `%` causes the remaining of the line to be treated as comment.

```
kaosProblem ::=
   domainDescription
   instantiationDirectiveList
   goalDeclaration+
[] domainDescription
   instantiationDirectiveList
   goalDeclaration+
   'resolution'
```

```
   goalDeclaration+
```

The overall structure of the file is a `kaosProblem`. It is made of a domain description, the instantiation directives and two lists of goals separated by the keyword `resolution`, possibly including resolution goals. The first list of goals defines the functional goals (LTL assertions) and claimed confidentiality goals. The second, facultative, list of goals defines the assumed confidentiality goals. There must be at least one claimed confidentiality goal.

```
domainDescription ::=
   [primitiveTypeDeclaration | objectDeclaration |
    dominvarDeclaration]*

dominvarDeclaration ::=
   'dominvar' assertion
```

The domain description contains the base type declaration, the object model declaration and the domain invariants. The domain invariants are handled the same way as functional goals.

A.1.1 Object Model and Base Types

Assertions are typed. Therefore, agent class and entity class are used as types. All used types must be declared before being used. The general principle is then to declare primitive types first, then entity and agent classes and finally associations.

```
primitiveTypeDeclaration ::=
   'basetype' "BaseTypeName"
[] 'enum' "EnumerationTypeName" '[' "EnumerationElement"+ ']'
```

Primitive types include enumerations and base types. Base types are similar to the ones of the Z specification language [PST91]. There are two build-in primitive types: integers (that is a base type) and booleans (that is an enumeration made of two values: `true` and `false`).

```
objectDeclaration ::=
```

```
   'entity' "EntityClassName" '{' attributeDefinitionList '}'
[] 'agent' "AgentClassName" '{' attributeDefinitionList '}'
[] 'association' "AssociationClassName" '{'
      participantDefinitionList '}'
```

There are three kinds of objects: *entities*, *agents* and *associations*. They all have attributes. Associations also have roles. Attributes and roles of associations are not syntactically distinguished. Therefore, they are grouped into the `participant` grammatical element. A participant can be either an attribute or a role, possibly with multiplicity.

```
attributeDefinitionList ::=
   attributeDefinition*

attributeDefinition ::=
   "AttributeName" ':' "AttributeType"
[] "AttributeName" ',' attributeDefinition
```

An agent or entity might have no attribute. The attribute type must be an already declared primitive type. It is forbidden to declare two attributes with the same names. When an object has several attributes of the same type, they can be declared together by listing them with a coma as separator.

```
participantDefinitionList ::=
   participantDefinition+

participantDefinition ::=
   "ParticipantName" ':' "ParticipantType"
[] "ParticipantName" ',' participantDefinition
[] "ParticipantName" multiplicity ':' "ParticipantType"
[] "ParticipantName" multiplicity ',' participantDefinition
```

Attributes and roles are not distinguished syntactically. Rather, their type distinguishes them. A participant whose type is an agent or an entity class is a role while a participant whose type is a primitive type is an attribute. Only roles might have a multiplicity. A role without multiplicity is assigned the default multiplicity $(0 : \infty)$.

The participant type must be an already declared primitive type, entity class or agent class. It is forbidden to declare two participants with the same names. When an association has several participant of the same type, they

can be declared together by listing them possibly with their multiplicity with a coma as separator.

```
multiplicity ::=
   '(' ':' ')'
[] '(' ':' 'integer' ')'
[] '(' 'integer' ':' ')'
[] '(' 'integer' ':' 'integer' ')'
```

A multiplicity is of the form *(min:max)*. Both *min* and *max* can be omitted. When the *min* is omitted, it is treated as 0. When the *max* is omitted is is treated as ∞. The *min* can be 0 or 1 and the *max* can be 1 or the keyword *"infinity"*. Other values are not supported.

The semantics of multiplicities is the one of OMT [BC95]. Multiplicity *(min:max)* of a role R of type T of association type A expresses that an agent or entity of type T must be implied in at least *min* and at most *max* associations A with role R.

A.1.2 Instantiation directives

```
instantiationDirectiveList ::=
   instantiationDirective*

instantiationDirective ::=
   "TypeName" ':' 'integer'
[] "TypeName" ':' '[' "InstanceName"* ']'
```

An instantiation directive declares the number of instance that must be created for a given primitive type, entity class, agent class or association class. It also assigns names to these instances. These names are used in the reporting of the scenario (except for association classes). An instantiation directive is of the form *"TypeName :"* followed by an integer or by a braced list of identificators.

In the case of an integer n, n instances are created and the names of these instances are built from the type name. For instance, the directive `Terminal:2` creates two instances of terminal with the names `Terminal0` and `Terminal1`. n cannot be zero and cannot be the keyword `infinity`.

Instantiation directives can be omitted. When a base type, or object type has no instantiation directive, it is assigned a default one and a warning is

reported by the tool.

A.1.3 Goals

```
goalDeclaration ::=
   'goal' 'goalQualifier' 'goalName' assertion
```

A goal is specified by the keyword `goal`, a qualifier, a goal name (a bracketed string) and an assertion. The legal qualifiers are: `confidentiality`, `achieve`, `avoid`, `maintain` and `cease`.

A.1.4 Assertions

This section gives the structure of assertions. In order to improve the readability, the structure of assertion is split into several subsections. Each subsection is given a name that relates to the semantics of the operators. Recall that assertions are typed and that agent classes and entity classes are used as types.

```
assertion ::=
   quantification
```

The tool does not allow free variables in assertions. Therefore, all variable used in an assertion must be quantified. Moreover, an assertion is always of type *boolean*.

A.1.4.1 Quantification

```
quantification ::=
 quantificationPrefix ':' quantification
[] '(' quantificationPrefix ')' quantification
[] entailment

quantificationPrefix ::=
 quantifier quantified +

quantifier ::=
   'exists'
```

```
[] 'forall'

quantified ::=
   "VariableName" ':' "TypeName"
[] "VariableName" ':' 'agent'
[] "VariableName" ',' quantified
```

A quantification can be written either with parenthesis or with colons (that is: `forall x:TypeName : P` or `(forall x:TypeName) P`). *TypeName* must be a declared type.

A quantification on agents or entities ranges over every instance of the agent or entity class declared by the instantiation directives. An extra domain is available in quantification, that is the *agent* domain. A quantification on agents ranges over every instance of every declared agent class.

When several quantified variable have the same type and the same quantifier, it is possible to regroup them into a coma-separated list.

There is a restriction on quantification on primitive types. In case of a universal quantification on a variable x with a primitive type, the quantified expression must be of the pattern $P \rightarrow Q$ or $P \Rightarrow Q$ where P forces x to a certain value. In case of an existential quantification on a variable x with a primitive type, the quantified expression must be of the pattern $P \vee Q$ where P and Q force x to a certain value.

An expression of the form $\forall$.. or $\exists$... evaluates to a Boolean. An expression that is prefixed by a quantifier must evaluate to a Boolean.

A.1.4.2 Temporal constructs

```
entailment ::=
   entailment '=>' temporal_unary
[] entailment '<=>' temporal_unary
[] temporal_unary

temporal_unary ::=
   '[]' temporal_unary
[] '<>' temporal_unary
[] '[*]' temporal_unary
[] '<*>' temporal_unary
[] 'o' temporal_unary
[] 'p' temporal_unary
```

```
[] implication
```

The supported temporal constructs are a subset of the classical operators for temporal referencing presented in Section 3.2.5.1 [MP92]. They are defined in Figure A.1.

ASCII representation	Operator/Connective
`[]`	□ (always in the future)
`<>`	◇ (some time in the future)
`[*]`	■ (always in the past)
`<*>`	⧫ (some time in the past)
`o`	○ (in the next state)
`p`	• (in the previous state)
`=>`	⇒ (entails)
`<=>`	⇔ (congruent)

Figure A.1: Temporal operators supported by CONCHITA

The application of these operators evaluate to Booleans. Their arguments must be Booleans.

A.1.4.3 Logic connectives

```
implication ::=
   implication '<->' disjunction
[] implication '->' disjunction
[] disjunction

disjunction ::=
   disjunction 'or' conjunction
[] conjunction

conjunction ::=
   conjunction 'and' negation
[] negation

negation ::=
   'not' negation
[] equality
```

The standard logical connectives are supported. They are defined in Figure A.2. Note that some connectives have several acceptable ASCII rep-

resentations.

ASCII representations	Operator/Connective
`&`, `and`	$\wedge$ (and)
`|`, `or`	$\vee$ (or)
`!`, `not`	$\neg$ (not)
`->`, `implies`	$\rightarrow$ (implication)
`<->`, `equivalent`	$\leftrightarrow$ (equivalence)

Figure A.2: Logical operators supported by CONCHITA

The application of these operators evaluate to Booleans. Their arguments must be Booleans.

A.1.4.4 Arithmetic operators

```
equality ::=
   equality '=' addition
[] equality '<' addition
[] equality '=<' addition
[] equality '>' addition
[] equality '>=' addition
[] addition
```

The application of these operators evaluate to Booleans. Their arguments must be integers except for the `=` operator that supports all primitive types, agent types or entity types. The two arguments of a `=` must be of the same type. This distinguishes the `=` operator from the `<->` operator.

```
addition ::=
   addition '+' multiplication
[] multiplication

multiplication ::=
   multiplication '*' atomic
[] atomic
```

The application of these operators evaluate to integers. Their arguments must be integers.

A.1.4.5 Atomic expressions

```
atomic ::=
   associationOccurence
[] confidentialityPattern
[] '(' quantification ')'
[] timeShiftedValue
```

A.1.4.6 Associations

```
associationOccurence ::=
   "AssociationName" '<'  associationMember+ '>'

associationMember ::=
   "ParticipantName" ':' atomic
```

`AssociationName` must be a declared association type. Every `ParticipantName` must be a declared participant name in the declaration of the association class. A participant name can occur at most once in the association members. When a participant is not represented in the members, it is taken as existentially quantified. For instance, `owning<owner:Somebody>` where association `owning` has another role `owned` is taken as:
`(exists Something:...) owning<owner:Somebody owned:Something>`)

The type of the expression tat appears in an association member (`atomic`) must be the equal to the declared type of the participant. There is an exception with variables quantified using the `agent` keyword. When the expression is one such variable, the type of the participant must only be a declared agent type. There is a limitation with the structure of `atomic`: this expression cannot be an association neither it can be a confidentiality construct.

A.1.4.7 Confidentiality Patterns

```
confidentialityPattern ::=
   'full-Confidential-now'
      '(' "AgentClassVariable" ',' confidentialExpression ')'
[] 'full-Confidential-forever'
      '(' "AgentClassVariable" ',' confidentialExpression ')'
[] 'full-Confidential-until'
```

```
'(' "AgentClassVariable" ',' confidentialExpression ','
      assertion ')'
```

`AgentClassVariable` must be a universally quantified variable of an agent class. It can also be universally quantified using the `agent` keyword.

Three confidentiality patterns are supported. Their syntax is defined in Figure A.3 where `A` represents an agent class variable, `x` represents a confidential expression and `E` represents an assertion. Their semantics is defined in Section 4.4.

ASCII representation	Pattern
`full-Confidential-now(A, x)`	full-Confidential$_A(x)$
`full-Confidential-forever(A, x)`	full-Confidential-forever$_A(x)$
`Cvu(A, x, E)`	full-Confidential-until$_{A,E}(x)$

Figure A.3: Confidentiality patterns supported by CONCHITA

```
confidentialExpression ::=
   "EntityAgentClassVariable" '.' "AttributeName"
[] "Variable"
[] associationuse '.' "ParticipantName"
```

The expression that is claimed/asumed to be confidential must be reducible into a single state variable. The model checker can reduce three kinds of expressions into a state variable:

- `"EntityAgentClassVariable" '.' "AttributeName"`: in this case, it is an attribute of some instance that is confidential.

  `EntityAgentClassVariable` must be a universally quantified variable of a declared entity or agent type and `AttributeName` must be a declared attribute of this agent or entity class.

- `"Variable"`: in this case, `Variable` must be a quantified variable of a primitive type. This allows fore more freedom in the specification of confidentiality goals. For instance,

  ```
  (forall x) Wedding<wife:Alice husband:Bob
     date:Variable> -> full-Confidential-forever(MrHide,x)
  ```

  with instantiation directive of `Wedding` set to 1 reduces to

```
Wedding0.holds and Wedding0.wife=Alice
and Wedding0.husband=Bob
-> full-Confidential-forever(MrHide,Wedding0.date)
```

- `associationuse '.' "ParticipantName"`: this is a shorthand notation that reduces to the participant of the association instances. For instance, the expression:

```
full-Confidential-forever(MrHide,
                      Wedding<wife:Alice
                              husband:Bob>.date)
```

reduces to

```
(forall x) Wedding<wife:Alice husband:Bob
   date:x> -> full-Confidential-forever(MrHide,x)
```

A.1.4.8 More Temporal Constructs

```
timeShiftedValue ::=
   timeShiftedValue '^'
[] terminalElement
```

The tool supports variable exponent a^- to denote the previous value of state variable a. Their semantics are described in Section 3.2.5.1 and their syntax is defined in Figure A.4. This exponents can only be applied on state variables.

The type of an expression of the form `E^` is the type of `E`. There is no restriction on the type of `E`.

ASCII representation	Operator
^	$^-$ (in the previous state)

Figure A.4: Temporal exponent supported by CONCHITA

A.1.4.9 Terminal Elements

```
TerminalElement ::=
   'integer'
[] 'init'
```

```
[] 'end'
[] "EnumerationElement"
[] "Variable"
[] "EntityAgentClassVariable" '.' "AttributeName"
```

The terminal elements are either constants (integer, init, end or enumeration element), quantified variables or attribute selection expressions.

Integers can be specified, notably for use in arithmetic expressions. The keyword `infinity` is forbidden. Also note that integers are always positive or null.

The keywords `init` and `end` are boolean constructs that evaluate to true at the first (resp last) state of the scenario and false at all other states.

An enumeration element is declared as a part of a enumeration. Its type is its enumeration.

Variables can be used as terminal elements. They must always be quantified as the tool does allow free variables (mainly because the type is specified in the quantification).

It is also possible to select attributes of agent or entities. `EntityAgentClassVariable` must be a quantified variable of an agent or entity type (but it cannot be quantified using the `agent` keyword because all agent might not have the same attributes). `AttributeName` must be a declared attribute of the agent or entity type.

A.2 Proving the Soundness of the Temporal Layer of CONCHITA

This section shows how temporal assertions are translated into non-temporal assertions in CONCHITA so as to enforce the two following properties:

$$\begin{aligned}&(\forall\ \sigma, P, \text{renaming})\\&\quad \text{renaming}(\sigma) \models \mathit{bound}(P, \mathit{strong}, |\sigma|, \text{renaming}, 1)\\&\qquad \rightarrow (\exists H)(\sigma H, 1) \models P \qquad \text{(A.1)}\\&(\forall\ \sigma, H, P, \text{renaming})\\&\quad (\sigma H, 1) \models P\\&\qquad \rightarrow \text{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \text{renaming}, 1) \qquad \text{(A.2)}\end{aligned}$$

Where:

- P is an assertion in LTL logics (see below for more definition about the grammar and semantics of this assertion)
- H is an infinite history (starting at position 1) on which P can be avaluated. Such an history is a function that defines the value of each proposition at each position of the time. Its signature is as follows:
  $temporal\ propositions \times \mathbb{N}_0 \rightsquigarrow \{T, F\}$
- σ is a finite history prefix of length $|\sigma|$. It is similar to infinite history except that it only ranges on a subset of $\mathbb{N}$ that is: $1, \ldots, |\sigma|$. Its signature is:
  $temporal\ propositions \times \{1, \ldots, |\sigma|\} \rightsquigarrow \{T, F\}$
- σH is an infinite history with prefix σ and continuation H. It is defined as follows:
  $\lambda P, t.\mathbf{if}\ t \leq |\sigma|\ \mathbf{then}\ \sigma(P, t)\mathbf{else}\ H(P, t - |\sigma|)\ \mathbf{fi}$
- "renaming" is a renaming of temporal propositions into non-temporal propositions. The signature of this function is:

  $$temporal\ propositions \times \mathbb{N}_0 \rightsquigarrow propositions$$

  It is a bijection that maps each proposition (of a supposed given model) at each position (of a supposed trace) onto a non-temporal proposition. By extention, it is also taken as function that maps a trace onto a non-temporal interpretation. For instance, renaming(σ) denotes a non-temporal interpretation such that, for all propositional symbol P:

  $$\text{renaming}(\sigma) =_{def} (\lambda P.\sigma(\text{renaming}^{-1}(P)))$$

- $\models$ is the "model" symbol that delivers the truth value of a temporal or propositional assertion on a given history or interpretation. This operator is overloaded and is used here for temporal and non-temporal models and assertions.
- "*bound*" is the translation function used in CONCHITA that translates a temporal assertion into a non-temporal assertion. This function has the following signature:

  $$temporal\ assertion \times \{strong, weak\} \times \mathbb{N} \times renaming \times \mathbb{N} \rightsquigarrow non - temporal\ assertion$$

Note: in the proofs of this section, we perform implicit skolemization of proof goals because they are always prefixed with universal quantifications. Moreover, we use the universally quantified variables as skolem constants.

A.2.1 Semantics of Temporal Assertions

This section defines the semantics of temporal assertions. The supported temporal operators are:

$\circ$	(in the next state)	$\bullet$	(in the previous state)
$\Diamond$	(some time in the future)	$\Box$	(always in the future)

The semantics of the above temporal operators is then defined as usual [MP92]. Beside this, we also define how these operators are to be interpreted on a finite trace.

A.2.1.1 Evaluating Temporal Assertions on Histories

Let be H an history and P a temporal assertion. The semantics of the temporal operators is defined as usual [MP92], for example:

$$\begin{aligned}
(H,i) &\models \circ P \quad \text{iff} \quad (H,i+1) \models P \\
(H,i) &\models \bullet P \quad \text{iff} \quad i \neq 1 \text{ and } (H,i-1) \models P \\
(H,i) &\models \Diamond P \quad \text{iff} \quad (H,j) \models P \text{ for some } j \geq i \\
(H,i) &\models \Box P \quad \text{iff} \quad (H,j) \models P \text{ for all } j \geq i
\end{aligned}$$

The logic connectives are also defined in this framework:

$$\begin{aligned}
(H,i) &\models \neg P \quad \text{iff} \quad \text{not } (H,i) \models P \\
(H,i) &\models P \wedge Q \quad \text{iff} \quad (H,i) \models P \text{ and } (H,i) \models Q \\
(H,i) &\models P \vee Q \quad \text{iff} \quad (H,i) \models P \text{ or } (H,i) \models Q
\end{aligned}$$

The terminal elements of this framework are the constants true, false and the proposition symbols, whose semantics are defined as follows:

$$\begin{aligned}
(H,i) &\models \text{true} \quad \text{always holds} \\
(H,i) &\models \text{false} \quad \text{never holds} \\
(H,i) &\models v \qquad \text{iff} \qquad H(v,i)
\end{aligned}$$

A.2.1.2 Evaluating Temporal Assertions on Finite Trace

We will also need to evaluate temporal assertions on finite traces in order to formulate one of our assumptions. On finite trace, it is often the case that the truth value of an assertion cannot be determined as true or false,

according to the semantics defined in the previous section. For instance, we cannot decide wether $\Box P$ evaluates to T or F on a finite trace where P holds at all states. All what we can say is that the assertion is not violated up to the end of the trace.

Partial evaluation of trace delivers a value from the set $\{\text{true}, \text{false}, \top\}$ where $\top$ captures the value of assertions whose value cannot be determined as true or false.

We use the $\models_d$ notation to denote evaluation of an assertion on a trace up to the position d of the trace. The value d is always taken as the length of the trace.

The semantics of the temporal operators is identical to the exact semantics except that it reasons on the tri-valued logics defined above and propositional symbols evaluate to $\top$ passed the end of the trace.

$$
\begin{array}{rcl}
(\sigma, i) \models_d \circ P & \rightsquigarrow & (\sigma, i+1) \models P \\
(\sigma, i) \models_d \bullet P & \rightsquigarrow & i \neq 1 \wedge_3 (\sigma, i-1) \models P \\
(\sigma, i) \models_d \Diamond P & \rightsquigarrow & \textbf{if } ((\sigma, j) \models_d P) = \text{true for some } j \geq i \\
 & & \quad \textbf{then } \text{true} \\
 & & \textbf{elseif } ((\sigma, j) \models_d P) = \text{false for all } j \geq i \\
 & & \quad \textbf{then } \text{false} \\
 & & \textbf{else } \top \textbf{ fi} \\
(\sigma, i) \models_d \Box P & \rightsquigarrow & \textbf{if } ((\sigma, j) \models_d P) = \text{true for all } j \geq i \\
 & & \quad \textbf{then } \text{true} \\
 & & \textbf{elseif } ((\sigma, j) \models_d P) = \text{false for some } j \geq i \\
 & & \quad \textbf{then } \text{false} \\
 & & \textbf{else } \top \textbf{ fi} \\
(\sigma, i) \models_d \neg P & \rightsquigarrow & \neg_3 (\sigma, i) \models P \\
(\sigma, i) \models_d P \wedge Q & \rightsquigarrow & ((\sigma, i) \models P) \wedge_3 ((\sigma, i) \models Q) \\
(\sigma, i) \models_d P \vee Q & \rightsquigarrow & ((\sigma, i) \models P) \vee_3 ((\sigma, i) \models Q) \\
(\sigma, i) \models_d \text{true} & \rightsquigarrow & \text{true} \\
(\sigma, i) \models_d \text{false} & \rightsquigarrow & \text{false} \\
(\sigma, i) \models_d v & \rightsquigarrow & \textbf{if } i > d \textbf{ then } \top \textbf{else } \sigma(v, i) \textbf{ fi}
\end{array}
$$

Note that our partial evaluation is not complete, that is: it is possible to find a trace and an assertion such that the assertion evaluates to $\top$ on the trace according to our partial evaluation though it might have been evaluated to true or false if we were reasoning about the models. For instance,

$(\Box P) \wedge (\Diamond \neg P)$ evaluates to false on all histories, trough our partial evaluation procedure delivers the value $\top$ on traces such that P is true at each positions.

A.2.2 Underlying assumptions

The proof of soundness of the temporal layer relies on two assumptions on P. The first is that P is divergence-free and the second is that it matches some grammar. Both assumptions are defined below.

A.2.2.1 Divergence-Free Assumption

The concept of divergence intuitively means that, in some requirements model, under some condition, an agent cannot enforce all the requirements he is responsible of. For instance, the following assertion contains a divergence:
$(P \Rightarrow \circ Q) \wedge (\Box \neg Q)$
The agent responsible of this is unable to enforce both parts of the assertion if P happens to be true. The concept of divergence has been defined in [vLDL98].

Put in other words, *"it is possible to derive an FSM that implements a divergence-free requirements model"*. This definition is too strong because it also implicitely contains reference to the fact that some variables can be monitored/controled by the FSM. These features are part of realizability assumption [Let01].

In our framework, we define divergence as follows: *"A divergence means that a trace that might be acceptable has no possible continuation"* In our framework, assertion P is divergence free if all trace for wich we cannot state that P does not hold have a possible continuation. This is formally defined as follows:

$$(\forall \sigma, P)\Big(\Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq \text{false} \rightarrow (\exists H)(\sigma H, 1) \models P\Big) \tag{A.3}$$

This definition is the reason why our partial evaluation is incomplete. If our partial evaluation were complete, this definition would be a tautology.

A.2.2.2 Grammar of Temporal Assertions:

The temporal assertions fed to the temporal layer of the tool match (or can be modified so as to match) the `TemporalAssertion` grammar defined below:

$$
\begin{aligned}
&\mathit{TemporalAssertion} ::= \\
&[] \ \Box \mathit{Assertion} \\
&[] \ \mathit{TemporalAssertion} \wedge \mathit{TemporalAssertion} \\
&[] \ \mathit{Assertion}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Assertion} ::= \\
&[] \ \bullet \mathit{Assertion} \\
&[] \ \circ \mathit{Assertion} \\
&[] \ \Diamond \mathit{Assertion} \\
&[] \ \mathit{Assertion} \wedge \mathit{Assertion} \\
&[] \ \mathit{Assertion} \vee \mathit{Assertion} \\
&[] \ \mathit{AtomicAssertion}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{AtomicAssertion} ::= \\
&[] \ \text{true} \\
&[] \ \text{false} \\
&[] \ \neg \mathit{AtomicAssertion} \\
&[] \ \mathit{Proposition}
\end{aligned}
$$

The consequence of this restricted grammar is that the following assertion holds:

$$
\begin{aligned}
&(\forall\ \sigma, P : \mathit{Assertion}, t) \\
&\quad t > |\sigma| \\
&\quad \wedge \Big((\sigma, t) \models_{|\sigma|} P \Big) = \text{false} \\
&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P \Big) = \text{false}
\end{aligned} \tag{A.4}
$$

This property is called *"trace self-representativity for Assertion"*. It is needed for proving the soundness of the bounding with strengtening. The trace self-representativity is proven in Section A.2.4.6. This restriction is not very restrictive because:

- Most requirements pattern fit into this grammar (cfr. the KAOS requirements patterns achieve, avoid, maintain, ...)

- all LTL assertion can be transformed into separate normal form that fit into this grammar. [FDP01]. An open issue for this model checker would be to explicitely generate snf, so as to remove the need for this grammar.

A.2.3 Starting the Proof

This section presents the first reduction of the proof goals (A.1) and (A.2). These goals are reduced by chaining with partial evaluation of temporal assertions on finite traces.

A.2.3.1 First Refinement of Time Bound with Strengthening

Proof goal (A.1) is decomposed into two subgoals (A.5) and (A.6) by chaining with partial evaluation of P on the trace σ as follows:

$$\begin{aligned}&(\forall\ \sigma, P : TemporalAssertion, \text{renaming})\\&\quad \text{renaming}(\sigma) \models bound(P, strong, |\sigma|, \text{renaming}, 1)\\&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq \text{false}\end{aligned} \tag{A.5}$$

$$\begin{aligned}&(\forall\ \sigma, P : TemporalAssertion, \text{renaming})\\&\quad \Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq \text{false}\\&\quad \rightarrow (\exists H)(\sigma H, 1) \models P\end{aligned} \tag{A.6}$$

Proof goal (A.5) is to be proven by induction on the grammar of Temporal Assertions. Proof goal (A.6) is identical to the divergence-free assumption (A.3). Therefore it is demonstrated.

A.2.3.2 First Refinement of Time Bound with Weakening

Proof goal (A.2) is decomposed into two subgoals (A.7) and (A.8) by chaining with partial evaluation of P on the trace σ as follows:

$$\begin{aligned}&(\forall\ \sigma, P : TemporalAssertion, \text{renaming})\\&\quad ((\sigma, 1) \models_{|\sigma|} P) \neq \text{false}\\&\quad \rightarrow \text{renaming}(\sigma) \models bound(P, weak, |\sigma|, \text{renaming}, 1)\end{aligned} \tag{A.7}$$

$$(\forall\ \sigma, H, P : TemporalAssertion)$$

$$(\sigma H, 1) \models P$$
$$\rightarrow ((\sigma, 1) \models_{|\sigma|} P) \neq \text{false} \tag{A.8}$$

Proof goal (A.7) is to be proven by induction on the grammar of Temporal Assertions. Proof goal (A.8) is a property of the partial evaluation of assertions on finite traces.

A.2.4 Bounding Time with Strengtening

This section defines the *bound* with strengtening on Temporal Assertion. It also proves the soundness of this rule. As already discussed, this proof relies on the trace self-representativity property. This property is also proven in this section.

A.2.4.1 Bounding Time for Temporal Assertions with Strengtening

Temporal assertions match the following grammar:

$$TemporalAssertion ::=$$
$$[]\ \Box Assertion$$
$$[]\ Assertion$$

This section defines the *bound* with strengtening on Temporal Assertion, up to the bounding of Assertions. It will also prove the soundness (A.5) of this rule, provided

1. the bounding operation with strengtening is sound on Assertions. This is defined as follows:

$$\begin{aligned}&(\forall\ \sigma, P : Assertion, \text{renaming}, t)\\ &\quad 1 \leq t \leq |\sigma|\\ &\quad \wedge \text{renaming}(\sigma) \models bound(P, strong, |\sigma|, \text{renaming}, t)\\ &\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \text{false}\end{aligned} \tag{A.9}$$

2. the *"trace self-representativity"* property, as defined in (A.4) holds.

The $\Box$ operator is bounded with strengtening by the *bound* function as follows:

$$\begin{aligned}&bound(\Box P, strong, |\sigma|, \text{renaming}, 1) =_{def}\\ &\qquad \wedge_{1 \leq i \leq |\sigma|} bound(P, strong, |\sigma|, \text{renaming}, i)\end{aligned} \tag{A.10}$$

Lemma 1 *Soundness of bounding of the always operator with strengtening:*

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming)\\
&\quad (\forall t'')\Big(1 \leq t \leq |\sigma|\\
&\qquad \wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t'')\\
&\qquad \rightarrow \Big((\sigma, t'') \models_{|\sigma|} P\Big) \neq false\Big)\\
&\quad \wedge (\forall t')\Big(t' > |\sigma| \wedge \Big((\sigma, t') \models_{|\sigma|} P\Big) = false\\
&\qquad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P\Big) = false\Big)\Big)\\
&\quad \wedge renaming(\sigma) \models bound(\Box P, strong, |\sigma|, renaming, 1)\\
&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} \Box P\Big) \neq false
\end{aligned}
\tag{A.11}
$$

Proof:

This proof proceeds by refutation. We assume that the consequent is false and show that the antecedent cannot be true. A case splitting is necessary.

Before turning the consequent to false and simplifying it, we simplify the last antecedent by expanding the bound function and according to the semantics of propositional logics into the following:

$$\wedge_{1 \leq i \leq |\sigma|} renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, i) \tag{A.12}$$

According to the bounded semantics of $\Box$ presented in Section A.2.1.1, the expression $(\sigma, 1) \models_{|\sigma|} \Box P$ appearing in the consequent of (A.11) evaluates to false if there is a time, represented here by the skolem constant s, such that $(\sigma, s) \models_{|\sigma|} P$ evaluates to false. Therefore, the negated consequent evaluates to (A.13).

$$\Big((\sigma, s) \models_{|\sigma|} P\Big) = false \tag{A.13}$$

We show now that the antecedent of (A.11) also evaluates to false. Two cases are envisionned on s: $s > |\sigma|$ and $1 \leq s \leq |\sigma|$.

- ***Case 1:*** $1 \leq s \leq |\sigma|$
  In this case, we can instantiate t'' in (A.11) to s, so that the antecedents of the first antecedent of (A.11) holds, according to (A.12). Therefore, the consequent of this second antecedent holds, leading to the following assertion:

  $$\Big((\sigma, s) \models_{|\sigma|} P\Big) \neq false \tag{A.14}$$

  Which contradicts with (A.13).

- ***Case 2:*** $s > |\sigma|$
  This case is more intricate because we need to use the second antecedent of (A.11).

  In this case, we can instantiate t' to s. With this instantiation, the antecedents of the second antecedent of (A.11) is true, according to (A.13) and to he case condition, leading to the following:

$$\Big((\sigma, 1) \models_{|\sigma|} P\Big) = \mathit{false} \tag{A.15}$$

  On the other hand, we instantiate t'' in (A.11) to 1, so that the antecedent of the first antecedent of (A.11) is true, according to (A.12). Therefore, the consequent of this second antecedent holds, leading to the following assertion:

$$\Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq \mathit{false} \tag{A.16}$$

  That contradicts with (A.15).

Q.E.D.

A.2.4.2 Soundness of Bounding Time for Temporal Assertions with Strengtening

The bounding operation with strengtening is sound for temporal assertions, as defined by (A.5), under the assumption that the bounding operation with strengtening is sound on Assertions as defined in (A.9).

Lemma 2 *Under the trace self representativity property and under the assumption that the bounding function is sound for assertions, it is also sound for temporal assertions. The latter is formally defined below:*

$$\begin{aligned}
&(\forall\ \sigma, P : TemporalAssertion, renaming) \\
&\quad renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, 1) \\
&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq \mathit{false}
\end{aligned} \tag{A.17}$$

Proof:

The proof is made of two cases matching the grammar on Temporal Assertions.

- ***Case 1:*** *P is of the pattern $\Box Q$ where Q is an Assertion*
  The proof goal reduces to:

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming)\\
&\quad (\forall t'')\Big(1 \leq t \leq |\sigma|\\
&\qquad \wedge renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t'')\\
&\qquad \rightarrow \Big((\sigma, t'') \models_{|\sigma|} Q\Big) \neq false\Big)\\
&\quad \wedge (\forall t')\Big(t' > |\sigma| \wedge \Big((\sigma, t') \models_{|\sigma|} Q\Big) = false\\
&\qquad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} Q\Big) = false\Big)\Big)\\
&\quad \wedge renaming(\sigma) \models bound(\Box Q, strong, |\sigma|, renaming, 1)\\
&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} \Box Q\Big) \neq false
\end{aligned}
\tag{A.18}
$$

  This is proven by Lemma 1.

- ***Case 2:*** *P is an Assertion.*
  The proof goal reduces to the following (the trace self-representativity has been removed for clarity)

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming)\\
&\quad (\forall t)\Big(1 \leq t \leq |\sigma|\\
&\qquad \wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t)\\
&\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false\Big)\\
&\quad \wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, 1)\\
&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq false
\end{aligned}
\tag{A.19}
$$

  This is trivial, we only need to instantiate t to 1.

Q.E.D.

A.2.4.3 Bounding Time for Assertions With Strengtening

Assertions match the following grammar:

$$
\begin{aligned}
&Assertion ::=\\
&[]\ \bullet Assertion\\
&[]\ \circ Assertion
\end{aligned}
$$

$$
\begin{aligned}
&[]\ \Diamond Assertion\\
&[]\ Assertion \wedge Assertion\\
&[]\ Assertion \vee Assertion\\
&[]\ AtomicAssertion
\end{aligned}
$$

The soundness property of the *bound* function on Assertion with strengtening is defined as follows:

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, \text{renaming}, t)\\
&\quad 1 \leq t \leq |\sigma|\\
&\quad \wedge \text{renaming}(\sigma) \models bound(P, strong, |\sigma|, \text{renaming}, t)\\
&\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \text{false}
\end{aligned}
\tag{A.20}
$$

This section defines the *bound* with strengtening on Assertions, up to the bounding of Atomic Assertions. It will also prove the soundness of this translation, provided the bounding operation is sound on Atomic Assertions. This is defined as follows:

$$
\begin{aligned}
&(\forall\ \sigma, P : AtomicAssertion, \text{renaming}, t)\\
&\quad 1 \leq t \leq |\sigma|\\
&\quad \wedge \text{renaming}(\sigma) \models bound(P, strong, |\sigma|, \text{renaming}, t)\\
&\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \text{false}
\end{aligned}
\tag{A.21}
$$

The $\circ$ operator is translated with strengtening by the *bound* function as follows:

$$
\begin{aligned}
&bound(\circ P, strong, |\sigma|, \text{renaming}, t) =_{def}\\
&\qquad \textbf{if } t \geq |\sigma| \textbf{ then}\\
&\qquad\quad \text{false}\\
&\qquad \textbf{else}\\
&\qquad\quad bound(P, strong, |\sigma|, \text{renaming}, t+1)\\
&\qquad \textbf{fi}
\end{aligned}
\tag{A.22}
$$

We can prove the soundness of this rule:

Lemma 3 *Soundness of bounding of the next operator with strengtening:*

$$
\begin{aligned}
&(\forall\ \sigma, P : AtomicAssertion, renaming, t)\\
&\quad 1 \leq t \leq |\sigma|\\
&\quad \wedge (\forall t') \Big(1 \leq t' \leq |\sigma|
\end{aligned}
$$

$$\begin{aligned}
&\wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t') \\
&\rightarrow \Big((\sigma, t') \models_{|\sigma|} P\Big) \neq false\Big) \\
\wedge renaming(\sigma) &\models bound(\circ P, strong, |\sigma|, renaming, t) \\
\rightarrow \Big((\sigma, t) &\models_{|\sigma|} \circ P\Big) \neq false
\end{aligned} \tag{A.23}$$

Proof:

The proof proceeds by two cases on t. Either it is equal to $|\sigma|$ or it is strictly lower than $|\sigma|$. the case of t being bigger than $|\sigma|$ is not considered because the assertion is trivial in this case.

- ***Case 1:*** $t = |\sigma|$
  In this case, the last antecedent of (A.23) can be reduced by applying the translation rule presented above into false, so that (A.23) trivially holds for that case.

- ***Case 2:*** $t < |\sigma|$
  In this case, the last antecedent of (A.23) can be reduced by applying the translation rule presented above into the following:

  $$renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t + 1) \tag{A.24}$$

  We can instantiate t' to $t + 1$ in the second antecedent of (A.23). All the premices of this instantiations are true, either becase of the case condition or because of (A.24). Therefore, the following is also true:

  $$\Big((\sigma, t + 1) \models_{|\sigma|} P\Big) \neq false \tag{A.25}$$

  On the other hand, under the case condition, the consequent of (A.23) can be reduced by applying the partial evaluation rules presented in Section A.2.1.1 into the following:

  $$\Big((\sigma, t + 1) \models_{|\sigma|} P\Big) \neq false \tag{A.26}$$

  That is trivially implied by (A.25)

Q.E.D.

The $\bullet$ operator is translated with strengtening by the *bound* function as follows:

$$bound(\bullet P, strong, |\sigma|, \text{renaming}, t) =_{def}$$

$$\begin{array}{l}\textbf{if } t = 1 \textbf{ then} \\ \quad \text{false} \\ \textbf{else} \\ \quad bound(P, strong, |\sigma|, \text{renaming}, t-1) \\ \textbf{fi}\end{array} \tag{A.27}$$

We can prove the soundness of this rule:

Lemma 4 *Soundness of bounding of the next operator with strengtening:*

$$\begin{array}{l}(\forall\ \sigma, P : AtomicAssertion, renaming, t) \\ \quad 1 \leq t \leq |\sigma| \\ \quad \wedge(\forall t')\Big(1 \leq t' \leq |\sigma| \\ \qquad \wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t') \\ \qquad \rightarrow \Big((\sigma, t') \models_{|\sigma|} P\Big) \neq false\Big) \\ \quad \wedge renaming(\sigma) \models bound(\bullet P, strong, |\sigma|, renaming, t) \\ \quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} \bullet P\Big) \neq false\end{array} \tag{A.28}$$

Proof:

The proof is trivial because the assertion produced by the bound function for this operator exactly captures its semantics. It proceeds by two cases on t, similarly to the previous proof.

- ***Case 1:*** $t = 1$
  In this case, the last antecedent of (A.28) can be reduced by applying the translation rule presented above into false, so that (A.28) trivially holds for that case.

- ***Case 2:*** $|\sigma| \geq t > 1$
  In this case, the last antecedent of (A.28) can be reduced by applying the translation rule presented above into the following:

  $$renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t-1) \tag{A.29}$$

  We can instantiate t' to $t-1$ in the second antecedent of (A.28). Under this instantiation, both antecedents of this antecedent are true, either because of the case condition or because of (A.29). Therefore, the following also holds:

  $$\Big((\sigma, t-1) \models_{|\sigma|} P\Big) \neq false \tag{A.30}$$

On the other hand, under the case condition, the consequent of (A.28) can be reduced by applying the partial evaluation rules presented in Section A.2.1.1 into the following:

$$\Big((\sigma, t-1) \models_{|\sigma|} P\Big) \neq \mathit{false} \tag{A.31}$$

That is trivially implied by (A.30)

Q.E.D.

The $\Diamond$ operator is translated with strengtening by the *bound* function as follows:

$$\begin{aligned} &\mathit{bound}(\Diamond P, \mathit{strong}, |\sigma|, \text{renaming}, t) =_{def} \\ &\quad \vee_{t \leq i \leq |\sigma|} \mathit{bound}(P, \mathit{strong}, |\sigma|, \text{renaming}, i) \end{aligned} \tag{A.32}$$

We can prove the soundness of this rule:

Lemma 5 *Soundness of bounding of the eventually operator with strengtening:*

$$\begin{aligned} &(\forall\ \sigma, P : \mathit{AtomicAssertion}, \mathit{renaming}, t) \\ &\quad 1 \leq t \leq |\sigma| \\ &\quad \wedge (\forall t')\Big(1 \leq t' \leq |\sigma| \\ &\qquad \wedge \mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{strong}, |\sigma|, \mathit{renaming}, t') \\ &\qquad \rightarrow \Big((\sigma, t') \models_{|\sigma|} P\Big) \neq \mathit{false}\Big) \\ &\quad \wedge \mathit{renaming}(\sigma) \models \mathit{bound}(\Diamond P, \mathit{strong}, |\sigma|, \mathit{renaming}, t) \\ &\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} \Diamond P\Big) \neq \mathit{false} \end{aligned} \tag{A.33}$$

Proof:

The last antecedent of (A.33) can be reduced by applying the bound function and according to the propositional logic semantics into (A.34).

$$\vee_{t \leq i \leq |\sigma|} \mathit{renaming}(\sigma) \models \mathit{bound}(\Diamond P, \mathit{strong}, |\sigma|, \mathit{renaming}, i) \tag{A.34}$$

Suppose that (A.34) is true because there is a j such that:

$$t \leq j \leq |\sigma| \tag{A.35}$$

$$\mathit{renaming}(\sigma) \models \mathit{bound}(\Diamond P, \mathit{strong}, |\sigma|, \mathit{renaming}, j) \tag{A.36}$$

We can instantiate t' to j in the second antecedent of (A.33). Under this instantiation, the two antecedents of this second antecedent are true, because of (A.35) and (A.36). therefore, the following is also true:

$$\Big((\sigma, j) \models_{|\sigma|} P\Big) \neq \mathit{false} \tag{A.37}$$

On the other hand, the consequent of (A.33) is true if the following is tue:

$$(\exists t')\Big(t' \geq t \wedge \Big((\sigma, t') \models_{|\sigma|} P\Big) \neq \mathit{false}\Big) \tag{A.38}$$

That is trivially true because it is true fot t' being instantiated to j and because of (A.35) and (A.37).

Q.E.D.

The $\wedge$ connector is translated with strengtening by the *bound* function as follows:

$$\begin{aligned} &\mathit{bound}(P \wedge Q, \mathit{strong}, |\sigma|, \text{renaming}, t) =_{\mathit{def}} \\ &\quad \mathit{bound}(P, \mathit{strong}, |\sigma|, \text{renaming}, t) \\ &\quad \wedge \mathit{bound}(Q, \mathit{strong}, |\sigma|, \text{renaming}, t) \end{aligned} \tag{A.39}$$

We can prove the soundness of this rule:

Lemma 6 *Soundness of bounding of conjunction with strengtening:*

$$\begin{aligned} &(\forall\ \sigma, P : \mathit{Assertion}, Q : \mathit{Assertion}, \mathit{renaming}, t) \\ &\quad 1 \leq t \leq |\sigma| \\ &\quad \wedge \mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{strong}, |\sigma|, \mathit{renaming}, t) \\ &\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \mathit{false} \\ &\quad \wedge \mathit{renaming}(\sigma) \models \mathit{bound}(Q, \mathit{strong}, |\sigma|, \mathit{renaming}, t) \\ &\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} Q\Big) \neq \mathit{false} \\ &\quad \wedge \mathit{renaming}(\sigma) \models \mathit{bound}(P \wedge Q, \mathit{strong}, |\sigma|, \mathit{renaming}, t) \\ &\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P \wedge Q\Big) \neq \mathit{false} \end{aligned} \tag{A.40}$$

Proof:

The proof proceeds by decomposing the consequent and the last antecedent and matching the results onto the second and third antecedent.

The last antecedent can be rewritten, by applying the translation rule presented above into the following:

$$\begin{aligned} renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t) \\ \wedge bound(Q, strong, |\sigma|, renaming, t) \end{aligned}$$

that can be reduced, according to the propositional semantics into the two following assertions:

$$renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t) \quad \text{(A.41)}$$

$$renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t) \quad \text{(A.42)}$$

The assertions (A.41) and (A.42) match the antecedents of the second an thirs antecedents of (A.40), so that the following assertions hold:

$$\Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false \quad \text{(A.43)}$$

$$\Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false \quad \text{(A.44)}$$

On the other hand, the assertion $(\sigma, t) \models_{|\sigma|} P \wedge Q$ that appears in the consequent of (A.40) can be reduced according to the bounded semantics of $\wedge$ presented in Section A.2.1.1 into the following:

$$\Big((\sigma, t) \models_{|\sigma|} P\Big) \wedge_3 \Big((\sigma, t) \models_{|\sigma|} Q\Big) \quad \text{(A.45)}$$

Accoding to the semantics of $\wedge_3$ and given assertions (A.43) and (A.44), assertion (A.45) evaluates eiter to true of $\top$, so that the consequent of (A.40) holds.

Q.E.D.

The $\vee$ connector is translated with strengtening by the *bound* function as follows:

$$\begin{aligned} bound(P \vee Q, strong, |\sigma|, \text{renaming}, t) =_{def} \\ bound(P, strong, |\sigma|, \text{renaming}, t) \\ \vee bound(Q, strong, |\sigma|, \text{renaming}, t) \end{aligned} \quad \text{(A.46)}$$

We can prove the soundness of this rule:

Lemma 7 *Soundness of bounding of disjunctions with strengtening:*

$$\begin{aligned} (\forall\ \sigma, P : Assertion, Q : Assertion, renaming, t) \\ 1 \leq t \leq |\sigma| \end{aligned}$$

$$\begin{aligned}
&\wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t) \\
&\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false \\
&\wedge renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t) \\
&\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} Q\Big) \neq false \\
&\wedge renaming(\sigma) \models bound(P \vee Q, strong, |\sigma|, renaming, t) \\
&\rightarrow \Big((\sigma, t) \models_{|\sigma|} P \vee Q\Big) \neq false
\end{aligned} \tag{A.47}$$

Proof:

The proof proceeds by decomposing the consequent and the last antecedent and matching the results onto the second and third antecedent. A case decomposition is needed.

The last antecedent can be rewritten, by applying the translation rule presented above into the following:

$$\begin{aligned}
renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t) \\
\vee bound(Q, strong, |\sigma|, renaming, t)
\end{aligned}$$

that can be reduced, according to the propositional semantics into the two following assertion:

$$\begin{aligned}
&\Big(renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t)\Big) \\
&\vee \Big(renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t)\Big)
\end{aligned} \tag{A.48}$$

As assertion (A.48) is supposed to hold, we envision two cases: either the first disjunct holds or the second one.

- ***Case 1:*** $renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t)$
  This case condition matches the antecedent of the second antecedent of (A.47), so that its consequent is supposed to hold. This leads to the following assertion:

  $$\Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false \tag{A.49}$$

  On the other hand, the assertion $(\sigma, t) \models_{|\sigma|} P \vee Q$ *that appears in the consequent of (A.47) can be reduced according to the bounded semantics of* $\vee$ *presented in Section A.2.1.1 into the following:*

  $$\Big((\sigma, t) \models_{|\sigma|} P\Big) \vee_3 \Big((\sigma, t) \models_{|\sigma|} Q\Big) \tag{A.50}$$

  Accoding to the semantics of $\vee_3$ *and assertion (A.49), assertion (A.50) evaluates eiter to true of* $\top$*, so that the consequent of (A.47) holds.*

- ***Case 2:*** $renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t)$
  The second case is identical to the first case with P being replaced by Q.

Q.E.D.

A.2.4.4 Soundness of Bounding Time for Assertions with Strengtening

Lemma 8 *Under the assumption that the bounding function is sound for atomic assertions, it is also sound for assertions. The latter is formally defined below:*

$$\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming, t)\\
&\quad 1 \leq t \leq |\sigma|\\
&\quad \wedge renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t)\\
&\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false
\end{aligned} \tag{A.51}$$

Proof:

The proof is made of six cases matching the grammar on Assertions.

- ***Case 1:*** *P is of the pattern $\circ Q$*
  The proof goal is then reduced with simplification into:

$$\begin{aligned}
&(\forall\ \sigma, Q : AtomicAssertion, renaming, t)\\
&\quad 1 \leq t \leq |\sigma|\\
&\quad \wedge (\forall t')\Big(1 \leq t' \leq |\sigma|\\
&\qquad\quad \wedge renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t')\\
&\qquad\quad \rightarrow \Big((\sigma, t') \models_{|\sigma|} Q\Big) \neq false\Big)\\
&\quad \wedge renaming(\sigma) \models bound(\circ Q, strong, |\sigma|, renaming, t)\\
&\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} \circ Q\Big) \neq false
\end{aligned} \tag{A.52}$$

This is proven by Lemma 3.

- ***Case 2:*** *P is of the pattern $\bullet Q$*
  The proof goal is then reduced with simplification into:

$$
\begin{array}{l}
(\forall\ \sigma, P : AtomicAssertion, renaming, t) \\
\quad 1 \leq t \leq |\sigma| \\
\quad \wedge (\forall t') \Big(1 \leq t' \leq |\sigma| \\
\qquad\qquad \wedge renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t') \\
\qquad\qquad \rightarrow \Big((\sigma, t') \models_{|\sigma|} Q \Big) \neq false \Big) \\
\quad \wedge renaming(\sigma) \models bound(\bullet Q, strong, |\sigma|, renaming, t) \\
\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} \bullet Q \Big) \neq false
\end{array}
\tag{A.53}
$$

  This is proven by Lemma 4.

- ***Case 3:*** *P is of the pattern $\Diamond Q$*
  The proof goal is then reduced with simplification into:

$$
\begin{array}{l}
(\forall\ \sigma, P : AtomicAssertion, renaming, t) \\
\quad 1 \leq t \leq |\sigma| \\
\quad \wedge (\forall t') \Big(1 \leq t' \leq |\sigma| \\
\qquad\qquad \wedge renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t') \\
\qquad\qquad \rightarrow \Big((\sigma, t') \models_{|\sigma|} Q \Big) \neq false \Big) \\
\quad \wedge renaming(\sigma) \models bound(\Diamond Q, strong, |\sigma|, renaming, t) \\
\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} \Diamond Q \Big) \neq false
\end{array}
\tag{A.54}
$$

  This is proven by Lemma 5.

- ***Case 4:*** *P is of the pattern $R \wedge S$*
  The proof goal is then reduced with simplification into:

$$
\begin{array}{l}
(\forall\ \sigma, R : Assertion, S : Assertion, renaming, t) \\
\quad 1 \leq t \leq |\sigma| \\
\quad \wedge renaming(\sigma) \models bound(R, strong, |\sigma|, renaming, t) \\
\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} R \Big) \neq false \\
\quad \wedge renaming(\sigma) \models bound(S, strong, |\sigma|, renaming, t) \\
\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} S \Big) \neq false \\
\quad \wedge renaming(\sigma) \models bound(R \wedge S, strong, |\sigma|, renaming, t) \\
\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} R \wedge S \Big) \neq false
\end{array}
\tag{A.55}
$$

  This is proven by Lemma 6

- ***Case 5:*** *P is of the pattern $R \vee S$*
  The proof goal is then reduced with simplification into:

$$\begin{aligned}
&(\forall\ \sigma, R : Assertion, S : Assertion, renaming, t) \\
&\quad 1 \leq t \leq |\sigma| \\
&\quad \wedge renaming(\sigma) \models bound(R, strong, |\sigma|, renaming, t) \\
&\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} R\Big) \neq false \\
&\quad \wedge renaming(\sigma) \models bound(S, strong, |\sigma|, renaming, t) \\
&\qquad\qquad \rightarrow \Big((\sigma, t) \models_{|\sigma|} S\Big) \neq false \\
&\quad \wedge renaming(\sigma) \models bound(R \vee S, strong, |\sigma|, renaming, t) \\
&\quad \rightarrow \Big((\sigma, t) \models_{|\sigma|} R \vee S\Big) \neq false
\end{aligned} \tag{A.56}$$

  This is proven by Lemma 7.

- ***Case 6:*** *P is an atomic assertion. (this is a base case)*
  The proof goal is then reduced into:

$$\begin{aligned}
&(\forall\ \sigma, P : AtomicAssertion, renaming) \\
&\quad renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, 1) \\
&\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P\Big) \neq false
\end{aligned} \tag{A.57}$$

  That is taken as assumption.

Q.E.D.

As far as they are evaluated within the trace, the bounding of atomic assertions is identical to the original atomic assertion. As a consequence, bounding an atomic assertion with strengthening or weakening does not influence the output, as far as it is evaluated withing the trace. Therefore, the strong/weak parameter is instantiated to "_" in this section.

The soundness property of the *bound* function on Atomic Assertion is defined as follows:

$$\begin{aligned}
&(\forall\ \sigma, P : Atomic\ Assertion, \text{renaming}, t) \\
&\quad 1 \leq t \leq |\sigma| \\
&\quad \rightarrow \Big(\text{renaming}(\sigma) \models bound(P, _, |\sigma|, \text{renaming}, t) \leftrightarrow (\sigma, t) \models_{|\sigma|} P\Big)
\end{aligned} \tag{A.58}$$

A consequence of this property is given below:

$$\begin{aligned}
&(\forall\ \sigma, P : Atomic\ Assertion, t) \\
&\qquad 1 \leq t \leq |\sigma| \rightarrow ((\sigma, t) \models_{|\sigma|} P) \in \{\text{true}; \text{false}\}
\end{aligned} \tag{A.59}$$

An atomic assertion is bounded by the *bound* function as follows:

$$\begin{aligned} bound(\text{true}, _, |\sigma|, \text{renaming}, t) &=_{def} \text{true} \\ bound(\text{false}, _, |\sigma|, \text{renaming}, t) &=_{def} \text{false} \\ bound(\neg P, _, |\sigma|, \text{renaming}, t) &=_{def} \neg bound(P, _, |\sigma|, \text{renaming}, t) \\ bound(v, _, |\sigma|, \text{renaming}, t) &=_{def} \text{renaming}(v, t) \end{aligned}$$

A.2.4.5 Soundness of the Time Bounding of Atomic Assertions

The soundness of atomic assertion, as defined in (A.58) is mostly trivial. The proof presented here therefore focus on the bounding of negation and of variables.

Lemma 9 *Soundness of the bound function on Atomic Assertions*

$$\begin{aligned} &(\forall\ \sigma, P : \mathit{Atomic\ Assertion}, \mathit{renaming}, t) \\ &\quad 1 \leq t \leq |\sigma| \\ &\quad \rightarrow \Big(\mathit{renaming}(\sigma) \models \mathit{bound}(P, _, |\sigma|, \mathit{renaming}, t) \\ &\qquad \leftrightarrow ((\sigma, t) \models_{|\sigma|} P) \neq \mathit{false}\Big) \end{aligned} \tag{A.60}$$

Proof:

The proof proceeds by structural induction over the grammar of atomic assertions.

- ***Case 1:*** *P is the constant true*
  This case is trivial and not proven here.
- ***Case 2:*** *P is the constant false*
  This case is trivial and not proven here.
- ***Case 3:*** *P is a negation*
  The proof goal reduces to the following:

$$\begin{aligned} &(\forall\ \sigma, Q : \mathit{Atomic\ Assertion}, \mathit{renaming}, t) \\ &\quad 1 \leq t \leq |\sigma| \\ &\quad \wedge\Big(\mathit{renaming}(\sigma) \models \mathit{bound}(Q, _, |\sigma|, \mathit{renaming}, t) \\ &\qquad \leftrightarrow ((\sigma, t) \models_{|\sigma|} Q) \neq \mathit{false}\Big) \end{aligned}$$

$$\rightarrow \Big(renaming(\sigma) \models bound(\neg Q, _, |\sigma|, renaming, t) \\ \leftrightarrow ((\sigma, t) \models_{|\sigma|} \neg Q) \neq false \Big) \quad (A.61)$$

By expanding the bound function in the consequent and according to the semantics of propositional logics, the proof goal can be rewritten as follows:

$$(\forall\ \sigma, Q : Atomic\ Assertion, renaming, t) \\ 1 \leq t \leq |\sigma| \\ \wedge \Big(renaming(\sigma) \models bound(Q, _, |\sigma|, renaming, t) \\ \leftrightarrow ((\sigma, t) \models_{|\sigma|} Q) \neq false \Big) \\ \rightarrow \Big(renaming(\sigma) \models bound(Q, _, |\sigma|, renaming, t) \\ \leftrightarrow ((\sigma, t) \models_{|\sigma|} \neg Q) = false \Big) \quad (A.62)$$

In the consequent, we can replace the antecedent with $((\sigma, t) \models_{|\sigma|} Q) \neq$ *false, according to the last antecedent. Therefore, we only have to prove the following:*

$$1 \leq t \leq |\sigma| \\ \rightarrow ((\sigma, t) \models_{|\sigma|} Q) \neq false \leftrightarrow ((\sigma, t) \models_{|\sigma|} \neg Q) = false \quad (A.63)$$

The latter is solved by a case on the value of $(\sigma, t) \models_{|\sigma|} Q)$.

- ***Case 1:*** $(\sigma, t) \models_{|\sigma|} Q) = true$
  According to the partial evaluation rule, $(\sigma, t) \models_{|\sigma|} \neg Q$ *evaluates then to false*
- ***Case 2:*** $(\sigma, t) \models_{|\sigma|} Q) = false$
  According to the partial evaluation rule, $(\sigma, t) \models_{|\sigma|} \neg Q$ *evaluates then to true*
- ***Case 3:*** $(\sigma, t) \models_{|\sigma|} Q) = \top$
  Due to (A.59), this case condition is false if $1 \leq t \leq |\sigma|$.

- ***Case 4:*** *P is a proposition symbol*
  In this case, by applying the partial evaluation rule and the bound function, the proof goal can be simplified into the following:

$$(\forall\ \sigma, P : Proposition\ Symbol, renaming, t) \\ 1 \leq t \leq |\sigma| \\ \rightarrow \Big(renaming(\sigma) \models renaming(P, t) \\ \leftrightarrow (\textbf{\textit{if}}\ t > |\sigma|\ \textbf{\textit{then}}\ \top \textbf{\textit{else}}\ \sigma(P, t)\ \textbf{\textit{fi}}) \neq false \Big) \quad (A.64)$$

That is simplified further, according to the semantics of porpositional logics and to the definition of the renaming function when applied on traces.

$$\begin{array}{l}(\forall\ \sigma, P : Proposition\ Symbol, renaming, t)\\ \quad 1 \leq t \leq |\sigma|\\ \quad \rightarrow \Big(\sigma(P,t) \leftrightarrow (\textbf{\textit{if}}\ t > |\sigma|\ \textbf{\textit{then}}\ \top \textbf{\textit{else}}\ \sigma(P,t)\ \textbf{\textit{fi}}) \neq false\Big)\end{array} \quad (A.65)$$

Under the antecedent, the condition appearing in the ***if*** *construct reduces to false, so that the proof goal is trivially true.*

Q.E.D.

A.2.4.6 Trace Self-Representativity for Assertions

The goal of this section is to prove the trace self-representativity for Assertions, as defined in (A.4). This proof is done by reasoning on the model first, and then by a structural induction on the grammar of assertions.

Lemma 10 *Cutting Down traces*
Let be s, a state, σ a finite trace and P an atomic assertion and t an integer. The two following assertions hold:

$$\begin{array}{l}t > 1\\ \wedge\Big((s\sigma, t) \models_{|\sigma|+1} P\Big) = false\\ \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} P\Big) = false\end{array} \quad (A.66)$$

$$\begin{array}{l}t > 1\\ \wedge\Big((s\sigma, t) \models_{|\sigma|+1} P\Big) = true\\ \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} P\Big) = true\end{array} \quad (A.67)$$

Proof:*Both assertions are proven together by structural induction on the grammar of atomic assertions.*

- ***Case 1:*** *P is the constant true*
  The proof goal is then:

$$\begin{array}{l}(\forall s, \sigma, t)\\ \quad t > 1\\ \quad \wedge Big((s\sigma, t) \models_{|\sigma|+1} true\Big) = false\\ \quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} true\Big) = false\end{array} \quad (A.68)$$

This is trivially true because the second antecedent is always false.

- ***Case 2:*** *P is the constant false*
  The proof goal is then:

$$\begin{aligned}&(\forall s, \sigma, t)\\&\quad t > 1\\&\quad \wedge Big((s\sigma, t) \models_{|\sigma|+1} false\Big) = false\\&\quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} false\Big) = false\end{aligned} \tag{A.69}$$

  This is trivially true because the consequent is always false.

- ***Case 3:*** *P is of the pattern* $\neg Q$
  The proof goal is then:

- ***Case 4:*** *P is a proposition symbol The proof goal is then:*

$$\begin{aligned}&(\forall s, \sigma, t, P : PropositionSymbol)\\&\quad t > 1\\&\quad \wedge\Big((s\sigma, t) \models_{|\sigma|+1} P\Big) = false\\&\quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} P\Big) = false\end{aligned} \tag{A.70}$$

  We split the proof of this into two cases on t*.*

  - ***Case 4.1:*** $t \geq |\sigma|$
    In this case, $(s\sigma, t) \models_{|\sigma|+1} P$ *evaluates to* $\top$*, so that the second antecedent of (A.70) reduces to false*
  - ***Case 4.2:*** $t < |\sigma|$
    In this case, $(s\sigma, t) \models_{|\sigma|+1} P$ *and* $(\sigma, t-1) \models_{|\sigma|} P$ *reduce to the same value, either true or false, so that (A.70) is proven.*

Q.E.D.

Lemma 11 *Cutting Down traces*
Let be s*, a state,* σ *a finite trace and* P *an assertion and* t *an integer,*

$$\begin{aligned}&t > 1\\&\wedge\Big((s\sigma, t) \models_{|\sigma|+1} P\Big) = false\\&\rightarrow \Big((\sigma, t-1) \models_{|\sigma|} P\Big) = false\end{aligned} \tag{A.71}$$

Proof:*This is proven by structural induction on the grammar of assertions.*

- ***Case 1:*** *P is of the pattern* $\bullet Q$
  The proof goal is then:

$$
\begin{aligned}
&(\forall s, \sigma, Q : Assertion, t) \\
&\quad (\forall t')\Big(t' > 1 \wedge \Big((s\sigma, t') \models_{|\sigma|+1} P\Big) = false \\
&\qquad \rightarrow \Big((\sigma, t' - 1) \models_{|\sigma|} P\Big) = false\Big) \\
&\quad \wedge t > 1 \\
&\quad \wedge \Big((s\sigma, t) \models_{|\sigma|+1} \bullet P\Big) = false \\
&\quad \rightarrow \Big((\sigma, t - 1) \models_{|\sigma|} \bullet P\Big) = false
\end{aligned} \tag{A.72}
$$

  This is proven by a case on the value of t*:* $t = 2$ *or* $t > 2$*.*

  - ***Case 1.1:*** $t = 2$
    In this case, the consequent of (A.72) is trivially true, because $\bullet P$ *evaluates to false at position* 1*.*
  - ***Case 1.2:*** $t > 2$
    In this case, the last antecedent and the consequent of (A.72) respectively reduce to (A.73) and (A.74).

$$\Big((s\sigma, t - 1) \models_{|\sigma|+1} P\Big) = false \tag{A.73}$$

$$\Big((\sigma, t - 2) \models_{|\sigma|} P\Big) = false \tag{A.74}$$

    We can instantiate t' *in the fist antecedent of (A.72) to* $t - 1$*, leaving the following:*

$$
\begin{aligned}
&\Big(t - 1 > 1 \wedge \Big((s\sigma, t - 1) \models_{|\sigma|+1} P\Big) = false \\
&\rightarrow \Big((\sigma, t - 2) \models_{|\sigma|} P\Big) = false\Big)
\end{aligned} \tag{A.75}
$$

    The two antecedents of (A.75) are true, because of the case condition and of (A.73), respectively, so that the consequent of (A.75) is also true. This consequent is equal to (A.74).

- ***Case 2:*** *P is of the pattern* $\circ Q$
  The proof goal is then:

$$
\begin{aligned}
&(\forall s, \sigma, Q : Assertion, t) \\
&\quad (\forall t')\Big(t' > 1 \wedge \Big((s\sigma, t') \models_{|\sigma|+1} P\Big) = false \\
&\qquad \rightarrow \Big((\sigma, t' - 1) \models_{|\sigma|} P\Big) = false\Big) \\
&\quad \wedge t > 1 \\
&\quad \wedge \Big((s\sigma, t) \models_{|\sigma|+1} \circ P\Big) = false \\
&\quad \rightarrow \Big((\sigma, t - 1) \models_{|\sigma|} \circ P\Big) = false
\end{aligned} \tag{A.76}
$$

The last antecedent and the consequent of (A.76) respectively reduce to (A.77) and (A.78).

$$\Big((s\sigma, t+1) \models_{|\sigma|+1} P\Big) = \mathit{false} \tag{A.77}$$

$$\Big((\sigma, t) \models_{|\sigma|} P\Big) = \mathit{false} \tag{A.78}$$

We can instantiate t' in the fist antecedent of (A.76) to $t+1$. This gives rize to the following:

$$\begin{aligned}&\Big(t+1 > 1 \wedge \Big((s\sigma, t+1) \models_{|\sigma|+1} P\Big) = \mathit{false}\\ &\rightarrow \Big((\sigma, t) \models_{|\sigma|} P\Big) = \mathit{false}\Big)\end{aligned} \tag{A.79}$$

The two antecedent of (A.79) hold, because they are either a tautology or because of (A.77), so that the consequent also hold. this consequent is equal to (A.78)

- ***Case 3:*** *P is of the pattern $\Diamond Q$*
  The proof goal is then:

$$\begin{aligned}&(\forall s, \sigma, Q : \mathit{Assertion}, t)\\ &\quad (\forall t')\Big(t' > 1 \wedge \Big((s\sigma, t') \models_{|\sigma|+1} P\Big) = \mathit{false}\\ &\qquad \rightarrow \Big((\sigma, t'-1) \models_{|\sigma|} P\Big) = \mathit{false}\Big)\\ &\quad \wedge t > 1\\ &\quad \wedge\Big((s\sigma, t) \models_{|\sigma|+1} \Diamond P\Big) = \mathit{false}\\ &\quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} \Diamond P\Big) = \mathit{false}\end{aligned} \tag{A.80}$$

According to the partial evaluation rule, the consequent of (A.80) reduces to:

$$(\forall k)\Big(k \geq t-1 \rightarrow ((\sigma, k) \models_{|\sigma|} P) = \mathit{false}\Big) \tag{A.81}$$

We skolemize (A.81) and use k as a skolem constant. Therefore, this assertion is reduced into the following assertion to be proven:

$$k \geq t-1 \rightarrow ((\sigma, k) \models_{|\sigma|} P) = \mathit{false} \tag{A.82}$$

We split the proof into two cases on k:

  - ***Case 1:*** *$k < t-1$*
    In this case, (A.82) is trivially true.

– ***Case 2:*** $k \geq t-1$
We reduce the last antecedent of (A.80) according to the partial evaluation rule into the following:

$$(\forall j)\Big(j \geq t \rightarrow ((s\sigma, j) \models_{|\sigma|+1} P) = \mathit{false}\Big) \tag{A.83}$$

We can instantiate j to $k+1$. This gives rize to the following:

$$k+1 \geq t \rightarrow ((s\sigma, k+1) \models_{|\sigma|+1} P) = \mathit{false} \tag{A.84}$$

According to the case condition, the premice is true, so that the consequent is also true. This leads to the following:

$$((s\sigma, k+1) \models_{|\sigma|+1} P) = \mathit{false} \tag{A.85}$$

We can instantiate t' to $k+1$ in the first antecedent of (A.80). This gives rize to the following:

$$\begin{aligned} & k+1 > 1 \wedge \Big((s\sigma, k+1) \models_{|\sigma|+1} P\Big) = \mathit{false} \\ & \rightarrow \Big((\sigma, k) \models_{|\sigma|} P\Big) = \mathit{false} \end{aligned} \tag{A.86}$$

The first antecedent of (A.86) is true because of the case condition and because of the second antecedent of (A.80). Similarly, the second antecedent of (A.86) is true, because of (A.85). Therefore, the consequent of (A.86) is also true. this consequent matches (A.82).

- ***Case 4:*** *P is of the pattern $Q \wedge R$*
The proof goal is then:

$$\begin{aligned} & (\forall s, \sigma, Q : \mathit{Assertion}, R : \mathit{Assertion}, t) \\ & \quad (\forall t')\Big(t' > 1 \wedge \Big((s\sigma, t') \models_{|\sigma|+1} Q\Big) = \mathit{false} \\ & \qquad\qquad \rightarrow \Big((\sigma, t'-1) \models_{|\sigma|} Q\Big) = \mathit{false}\Big) \\ & \quad \wedge (\forall t'')\Big(t'' > 1 \wedge \Big((s\sigma, t'') \models_{|\sigma|+1} R\Big) = \mathit{false} \\ & \qquad\qquad \rightarrow \Big((\sigma, t''-1) \models_{|\sigma|} R\Big) = \mathit{false}\Big) \\ & \quad \wedge t > 1 \\ & \quad \wedge \Big((s\sigma, t) \models_{|\sigma|+1} Q \wedge R\Big) = \mathit{false} \\ & \quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} Q \wedge R\Big) = \mathit{false} \end{aligned} \tag{A.87}$$

The last antecedent can be reduced by applying the partial evaluation rule into the following:

$$\Big((s\sigma, t) \models_{|\sigma|+1} Q\Big) = \mathit{false} \vee \Big((s\sigma, t) \models_{|\sigma|+1} R\Big) = \mathit{false} \tag{A.88}$$

In order to finish the proof, we envision two cases about (A.88). Either the first or the secund disjunct is true.

- ***Case 4.1:*** $\Big((s\sigma, t) \models_{|\sigma|+1} Q\Big) = \mathit{false}$
  In this case, we can instantiate t' to t in the first antecedent of (A.87), leading to the following:

$$\begin{aligned} & t > 1 \wedge \Big((s\sigma, t) \models_{|\sigma|+1} Q\Big) = \mathit{false} \\ & \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} Q\Big) = \mathit{false} \end{aligned} \tag{A.89}$$

  Both antecedents of (A.89) are true because of the case condition and because of the third premice of (A.87), so that the following also holds:

$$\Big((\sigma, t-1) \models_{|\sigma|} Q\Big) = \mathit{false} \tag{A.90}$$

  The consequent of (A.87) is true because of (A.90)

- ***Case 4.2:*** $\Big((s\sigma, t) \models_{|\sigma|+1} R\Big) = \mathit{false}$
  This case is similar to the previous one, with Q being replaced by R.

- ***Case 5:*** *P is of the pattern $Q \vee R$*
  The proof goal is then:

$$\begin{aligned} & (\forall s, \sigma, Q : \mathit{Assertion}, R : \mathit{Assertion}, t) \\ & \quad (\forall t')\Big(t' > 1 \wedge \Big((s\sigma, t') \models_{|\sigma|+1} Q\Big) = \mathit{false} \\ & \qquad \rightarrow \Big((\sigma, t'-1) \models_{|\sigma|} Q\Big) = \mathit{false}\Big) \\ & \quad \wedge (\forall t'')\Big(t'' > 1 \wedge \Big((s\sigma, t'') \models_{|\sigma|+1} R\Big) = \mathit{false} \\ & \qquad \rightarrow \Big((\sigma, t''-1) \models_{|\sigma|} R\Big) = \mathit{false}\Big) \\ & \quad \wedge t > 1 \\ & \quad \wedge \Big((s\sigma, t) \models_{|\sigma|+1} Q \vee R\Big) = \mathit{false} \\ & \quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} Q \vee R\Big) = \mathit{false} \end{aligned} \tag{A.91}$$

  The last antecedent of (A.91) can be simplified into the two following assertions:

$$\Big((s\sigma, t) \models_{|\sigma|+1} Q\Big) = \mathit{false} \tag{A.92}$$

$$\Big((s\sigma, t) \models_{|\sigma|+1} R\Big) = \mathit{false} \tag{A.93}$$

Similarly, the consequent can be simplified into the two following assertions:

$$\Big((\sigma, t-1) \models_{|\sigma|} Q\Big) = \mathit{false} \tag{A.94}$$

$$\Big((\sigma, t-1) \models_{|\sigma|} R\Big) = \mathit{false} \tag{A.95}$$

Both of these assertions must be proven; however, we will only show here how to prove (A.94). This is done by instantiating t' to t in the first antecedent of (A.91). This gives rise to the following:

$$\begin{aligned} &\Big(t > 1 \wedge \Big((s\sigma, t) \models_{|\sigma|+1} Q\Big) = \mathit{false} \\ &\rightarrow \Big((\sigma, t-1) \models_{|\sigma|} Q\Big) = \mathit{false}\Big) \end{aligned} \tag{A.96}$$

The two antecedents of (A.96) are true because of the third antecedent of (A.91) and because of (A.92), respectively, so that the consequent is also true. This proves (A.94).

- ***Case 6:*** *P is an atomic assertion*
  The proof goal is then:

$$\begin{aligned} &(\forall s, \sigma, P : \mathit{AtomicAssertion}, t) \\ &\quad t > 1 \\ &\quad \wedge\Big((s\sigma, t) \models_{|\sigma|+1} P\Big) = \mathit{false} \\ &\quad \rightarrow \Big((\sigma, t-1) \models_{|\sigma|} P\Big) = \mathit{false} \end{aligned} \tag{A.97}$$

  This is subsumed by Lemma 10.

Q.E.D.

Lemma 12 *Extending Traces*

$$\begin{aligned} &(\forall s, \sigma, t, P : \mathit{Assertion}) \\ &\quad t > 1 \\ &\quad \wedge\Big((\sigma, t) \models_{|\sigma|} P\Big) = \mathit{false} \\ &\quad \rightarrow \Big((\sigma s, t) \models_{|\sigma|+1} P\Big) = \mathit{false} \end{aligned} \tag{A.98}$$

Theorem 4 *Trace Self-representativity for assertions*

$$\begin{aligned} &(\forall \sigma, t, P : \mathit{Assertion}) \\ &\quad t > |\sigma| \\ &\quad \wedge\Big((\sigma, t) \models_{|\sigma|} P\Big) = \mathit{false} \\ &\quad \rightarrow \Big((\sigma, 1) \models_{|\sigma|} P\Big) = \mathit{false} \end{aligned} \tag{A.99}$$

This is proven by building the second member from the first one as follows:

- ***Step 1*** *Extend* σ
  We propose an extended trace $(\sigma\tau\sigma)$ *such that the second sigma starts at time* t*, that is:* $(|\sigma\tau| = t-1)$*. Note that* τ *might possibly be a trace of length zero.*

  From the premice of (A.99) we get the following, due to the application of Lemma 12 for each of the states in $\tau\sigma$*:*

  $$\Big((\sigma\tau\sigma, t) \models_{|\sigma|} P\Big) = \mathit{false} \tag{A.100}$$

- ***Step 2*** *Cut down* $\sigma\tau\sigma$ *to* σ *by removing the first states.*
  From (A.100), we get the following, due to the application of Lemma 11 once per removed state in the trace:

  $$\Big((\sigma, t - |\sigma\tau|) \models_{|\sigma|} P\Big) = \mathit{false} \tag{A.101}$$

  as $|\sigma\tau| = t-1$*, this simplifies to:*

  $$\Big((\sigma, 1) \models_{|\sigma|} P\Big) = \mathit{false} \tag{A.102}$$

  That is the consequent of (A.99).

Q.E.D.

A.2.5 Bounding Time With Weakening

This section defines the *bound* with weakening on Temporal Assertion. It also proves the soundness of this rule.

A.2.5.1 Bounding Time for Temporal Assertions with Weakening

Temporal assertions match the following grammar:

$$\begin{aligned} &\mathit{TemporalAssertion} ::= \\ &[]\ \Box \mathit{Assertion} \\ &[]\ \mathit{Assertion} \end{aligned}$$

This section defines the *bound* with weakening on Temporal Assertion, up to the bounding of Assertions. It will also prove the soundness (A.7) of this rule, provided the bounding operation with weakening is sound on Assertions. This is defined as follows:

$$
\begin{aligned}
&(\forall\ \sigma, P : Temporal\ Assertion, \text{renaming}) \\
&\quad ((\sigma, 1) \models_{|\sigma|} P) \neq \text{false} \\
&\quad \rightarrow \text{renaming}(\sigma) \models bound(P, weak, |\sigma|, \text{renaming}, 1) \qquad \text{(A.103)} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(A.104)}
\end{aligned}
$$

The $\Box$ operator is bounded with weakening by the *bound* function as follows:

$$
\begin{aligned}
&bound(\Box P, weak, |\sigma|, \text{renaming}, 1) =_{def} \\
&\qquad \wedge_{1 \leq i \leq |\sigma|} bound(P, weak, |\sigma|, \text{renaming}, i) \qquad \text{(A.105)}
\end{aligned}
$$

Lemma 13 *Soundness of bounding of the always operator with weakening:*

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming) \\
&\quad (\forall t'') \Big(1 \leq t'' \leq |\sigma| \wedge \Big((\sigma, t'') \models_{|\sigma|} P \Big) \neq false \\
&\qquad\qquad \rightarrow renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t'') \Big) \\
&\quad \wedge \Big((\sigma, 1) \models_{|\sigma|} \Box P \Big) \neq false \\
&\quad \rightarrow renaming(\sigma) \models bound(\Box P, strong, |\sigma|, renaming, 1) \qquad \text{(A.106)}
\end{aligned}
$$

Proof:

This proof proceeds by refutation. We suppose that the consequent is false and show that the antecedent is also false. The consequent of (A.106) can be simplified by expanding the bound function and according to the semantics of popositional logics into the following:

$$
\wedge_{1 \leq i \leq |\sigma|} renaming(\sigma) \models bound(P, weak, |\sigma|, renaming, i) \qquad \text{(A.107)}
$$

As we suppose that (A.107) is false, it means that for some j, the two following assertions are true:

$$
\begin{aligned}
&1 \leq j \leq |\sigma| \qquad \text{(A.108)} \\
&\neg \Big(renaming(\sigma) \models bound(P, weak, |\sigma|, renaming, j) \Big) \qquad \text{(A.109)}
\end{aligned}
$$

In the first premice of (A.106), we isntantiate t'' to j. Under this instantiation, the first antecedent if true, because of (A.108) and the consequent is false, because of (A.109). Therefore, the following assertion is true:

$$\Big((\sigma, j) \models_{|\sigma|} P\Big) = \mathit{false} \tag{A.110}$$

On the other hand, the last antecedent of (A.106) simplifies according to the partial evaluation rule into:

$$(\forall t)\Big(\Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \mathit{false}\Big) \tag{A.111}$$

Which turns out to be false because the quantified assertion is known to be false for t being instantiated to j, according to (A.110).

Q.E.D.

A.2.5.2 Soundness of Bounding Time for Temporal Assertions with Weakening

The bounding operation with weakening is sound for temporal assertions, as defined by (A.7), under the assumption that the bounding operation with weakening is sound on Assertions as defined in (A.103).

Lemma 14 *Under the assumption that the bounding function is sound for assertions, it is also sound for temporal assertions. The latter is formally defined below:*

$$\begin{aligned}&(\forall\ \sigma, P : \mathit{Assertion}, \mathit{renaming})\\ &\quad((\sigma, 1) \models_{|\sigma|} P) \neq \mathit{false}\\ &\quad\rightarrow \mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathit{renaming}, 1)\end{aligned} \tag{A.112}$$

Proof:

The proof is made of two cases matching the grammar on Temporal Assertions.

- ***Case 1:*** *P is of the pattern $\Box Q$ where Q is an Assertion*
  The proof goal reduces to:

$$\begin{aligned}&(\forall\ \sigma, Q : \mathit{Assertion}, \mathit{renaming})\\ &\quad(\forall t'')\Big(1 \leq t'' \leq |\sigma| \wedge \Big((\sigma, t'') \models_{|\sigma|} Q\Big) \neq \mathit{false}\end{aligned}$$

$$\rightarrow renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t'')\Big)$$
$$\wedge\Big((\sigma, 1) \models_{|\sigma|} \Box Q\Big) \neq false$$
$$\rightarrow renaming(\sigma) \models bound(\Box Q, strong, |\sigma|, renaming, 1) \quad \text{(A.113)}$$

This is proven by Lemma 13.

- ***Case 2:*** *P is an Assertion.*
  The proof goal reduces to the following:

$$(\forall\ \sigma, Q : Assertion, renaming)$$
$$(\forall t)\Big(1 \leq t \leq |\sigma| \wedge ((\sigma, t) \models_{|\sigma|} Q) \neq false$$
$$\rightarrow renaming(\sigma) \models bound(Q, weak, |\sigma|, renaming, t)\Big)$$
$$\wedge((\sigma, 1) \models_{|\sigma|} Q) \neq false$$
$$\rightarrow renaming(\sigma) \models bound(Q, weak, |\sigma|, renaming, 1) \quad \text{(A.114)}$$

This is trivial, we only need to instantiate t to 1 *in (A.19)*

Q.E.D.

A.2.5.3 Bounding Time for Assertions With Weakening

Assertions match the following grammar:

$$Assertion ::=$$
$$[]\ \bullet Assertion$$
$$[]\ \circ Assertion$$
$$[]\ \Diamond Assertion$$
$$[]\ Assertion \wedge Assertion$$
$$[]\ Assertion \vee Assertion$$
$$[]\ AtomicAssertion$$

The soundness property of the *bound* function on Assertion with weakening is defined as follows:

$$(\forall\ \sigma, P : Assertion, \text{renaming}, t)$$
$$1 \leq t \leq |\sigma|$$
$$\wedge\Big((\sigma, t) \models_{|\sigma|} Q\Big) \neq \text{false}$$
$$\rightarrow \text{renaming}(\sigma) \models bound(Q, strong, |\sigma|, \text{renaming}, t) \quad \text{(A.115)}$$

This section defines the *bound* with weakening on Assertions, up to the bounding of Atomic Assertions. It will also prove the soundness of this translation, provided the bounding operation is sound on Atomic Assertions. This is defined as follows:

$$\begin{aligned}&(\forall\ \sigma, P : \mathit{Atomic\ Assertion}, \text{renaming}, t)\\ &\quad 1 \leq t \leq |\sigma|\\ &\quad \wedge\Big((\sigma, t) \models_{|\sigma|} Q\Big) \neq \text{false}\\ &\quad \rightarrow \text{renaming}(\sigma) \models \mathit{bound}(Q, \mathit{strong}, |\sigma|, \text{renaming}, t)\end{aligned} \tag{A.116}$$

The $\bullet$ operator is translated with weakening by the *bound* function as follows:

$$\begin{aligned}&\mathit{bound}(\bullet P, \mathit{weak}, |\sigma|, \text{renaming}, t) =_{def}\\ &\qquad \textbf{if } t = 0 \textbf{ then}\\ &\qquad\quad \text{false}\\ &\qquad \textbf{else}\\ &\qquad\quad \mathit{bound}(P, \mathit{strong}, |\sigma|, \text{renaming}, t-1)\\ &\qquad \textbf{fi}\end{aligned} \tag{A.117}$$

We can prove the soundness of this rule:

Lemma 15 *Soundness of bounding of the "prev" operator with weakening:*

$$\begin{aligned}&(\forall\ \sigma, P : \mathit{Assertion}, \mathit{renaming}, t)\\ &\quad (\forall t')\Big(1 \leq t' \leq |\sigma| \wedge \Big((\sigma, t') \models_{|\sigma|} P\Big) \neq \mathit{false}\\ &\qquad\quad \rightarrow \mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{strong}, |\sigma|, \mathit{renaming}, t')\Big)\\ &\quad \wedge 1 \leq t \leq |\sigma|\\ &\quad \wedge\Big((\sigma, t) \models_{|\sigma|} \bullet P\Big) \neq \mathit{false}\\ &\quad \rightarrow \mathit{renaming}(\sigma) \models \mathit{bound}(\bullet P, \mathit{strong}, |\sigma|, \mathit{renaming}, t)\end{aligned} \tag{A.118}$$

Proof:

This proof is very similar to the one of Lemma 4 and is skipped.

Q.E.D.

The $\circ$ operator is translated with weakening by the *bound* function as follows:

$$\mathit{bound}(\circ P, \mathit{weak}, |\sigma|, \text{renaming}, t) =_{def}$$

$$\begin{array}{l}\textbf{if } t \geq |\sigma| \textbf{ then} \\ \quad \text{true} \\ \textbf{else} \\ \quad bound(P, strong, |\sigma|, \text{renaming}, t+1) \\ \textbf{fi}\end{array} \tag{A.119}$$

We can prove the soundness of this rule:

Lemma 16 *Soundness of bounding of the "next" operator with weakening:*

$$\begin{array}{l}(\forall\ \sigma, P : Assertion, renaming, t) \\ \quad (\forall t')\Big(1 \leq t' \leq |\sigma| \wedge \Big((\sigma, t') \models_{|\sigma|} P\Big) \neq false \\ \qquad \rightarrow renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t')\Big) \\ \quad \wedge 1 \leq t \leq |\sigma| \\ \quad \wedge\Big((\sigma, t) \models_{|\sigma|} \circ P\Big) \neq false \\ \quad \rightarrow renaming(\sigma) \models bound(\circ P, weak, |\sigma|, renaming, t)\end{array} \tag{A.120}$$

Proof:

This proof proceeds by a case splitting on the value of t

- ***Case 1:*** $t \geq |\sigma|$
  In this case, the consequent of (A.120) reduces to true by expanding the bound function.

- ***Case 2:*** $1 \leq t < |\sigma|$
  In this case, the consequent of (A.120) can be rewritten by expanding the bound function into the following:

  $$renaming(\sigma) \models bound(P, weak, |\sigma|, renaming, t+1) \tag{A.121}$$

  On the other hand, the last antecedent can be rewritten into the following according to the partial evaluation rule:

  $$\Big((\sigma, t+1) \models_{|\sigma|} P\Big) \neq false \tag{A.122}$$

  We instantiate t' in the first antecedent of (A.120) to $t+1$. Under this instantiation, the two antecedents turn out to be true, either because of the case condition or because of (A.122). This give rize to the following assertion:

  $$renaming(\sigma) \models bound(P, weak, |\sigma|, renaming, t+1) \tag{A.123}$$

  That is identical to (A.121).

Q.E.D.

The $\Diamond$ operator is translated with weakening by the *bound* function as follows:

$$bound(\Diamond P, weak, |\sigma|, \text{renaming}, t) =_{def} \text{true} \tag{A.124}$$

We can prove the soundness of this rule:

Lemma 17 *Soundness of bounding of the "eventually" operator with weakening:*

$$\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming, t) \\
&\quad \wedge 1 \leq t \leq |\sigma| \\
&\quad \wedge \Big((\sigma, t) \models_{|\sigma|} \circ P \Big) \neq false \\
&\quad \rightarrow renaming(\sigma) \models bound(\circ P, weak, |\sigma|, renaming, t)
\end{aligned} \tag{A.125}$$

Proof:

The consequent of (A.125) reduces to true, therefore the lemma is trivially proven.

The $\wedge$ connector is translated with weakening by the *bound* function as follows:

$$\begin{aligned}
&bound(P \wedge Q, weak, |\sigma|, \text{renaming}, t) =_{def} \\
&\quad bound(P, weak, |\sigma|, \text{renaming}, t) \wedge bound(Q, weak, |\sigma|, \text{renaming}, t)
\end{aligned} \tag{A.126}$$

We can prove the soundness of this rule:

Lemma 18 *Soundness of bounding of conjunction with weakening:*

$$\begin{aligned}
&(\forall\ \sigma, P : Assertion, Q : Assertion, renaming, t) \\
&\quad 1 \leq t \leq |\sigma| \\
&\quad \wedge \Big((\sigma, t) \models_{|\sigma|} P \Big) \neq false \\
&\qquad \rightarrow renaming(\sigma) \models bound(P, weak, |\sigma|, renaming, t) \\
&\quad \wedge \Big((\sigma, t) \models_{|\sigma|} Q \Big) \neq false \\
&\qquad \rightarrow renaming(\sigma) \models bound(Q, weak, |\sigma|, renaming, t) \\
&\quad \wedge \Big((\sigma, t) \models_{|\sigma|} P \wedge Q \Big) \neq false \\
&\quad \rightarrow renaming(\sigma) \models bound(P \wedge Q, weak, |\sigma|, renaming, t)
\end{aligned} \tag{A.127}$$

Proof:

The proof proceeds by decomposing the consequent and the last antecedent and matching the results onto the second and third antecedent.

The last antecedent can be rewritten, according to the partial evaluation rule into the two following assertions:

$$\Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \mathit{false} \tag{A.128}$$

$$\Big((\sigma, t) \models_{|\sigma|} Q\Big) \neq \mathit{false} \tag{A.129}$$

The assertions (A.128) and (A.129) match the antecedents of the second an third antecedents of (A.127), so that the following assertions hold:

$$\mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \tag{A.130}$$

$$\mathit{renaming}(\sigma) \models \mathit{bound}(Q, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \tag{A.131}$$

On the other hand, the consequent of (A.127) can be reduced by expanding the bound function into the following:

$$\begin{aligned} &\mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \\ &\wedge \mathit{renaming}(\sigma) \models \mathit{bound}(Q, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \end{aligned} \tag{A.132}$$

That is trivially true because of (A.130) and (A.131).

Q.E.D.

The $\vee$ connector is translated with weakening by the *bound* function as follows:

$$\begin{aligned} &\mathit{bound}(P \vee Q, \mathit{weak}, |\sigma|, \mathrm{renaming}, t) =_{def} \\ &\quad \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathrm{renaming}, t) \vee \mathit{bound}(Q, \mathit{weak}, |\sigma|, \mathrm{renaming}, t) \end{aligned} \tag{A.133}$$

We can prove the soundness of this rule:

Lemma 19 *Soundness of bounding of disjunctions with strengtening:*

$$\begin{aligned} &(\forall\ \sigma, P : \mathit{Assertion}, Q : \mathit{Assertion}, \mathit{renaming}, t) \\ &\quad 1 \leq t \leq |\sigma| \\ &\quad \wedge \Big((\sigma, t) \models_{|\sigma|} P\Big) \neq \mathit{false} \\ &\qquad \rightarrow \mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \\ &\quad \wedge \Big((\sigma, t) \models_{|\sigma|} Q\Big) \neq \mathit{false} \\ &\qquad \rightarrow \mathit{renaming}(\sigma) \models \mathit{bound}(Q, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \\ &\quad \wedge \Big((\sigma, t) \models_{|\sigma|} P \vee Q\Big) \neq \mathit{false} \\ &\quad \rightarrow \mathit{renaming}(\sigma) \models \mathit{bound}(P \vee Q, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \end{aligned} \tag{A.134}$$

Proof:

The proof proceeds by decomposing the consequent and the last antecedent and matching the results onto the second and third antecedent. A case decomposition is needed.

The last antecedent can be rewritten, according to the partial evaluation rule, into the following:

$$\Big(\Big((\sigma,t) \models_{|\sigma|} P\Big) \vee_3 \Big((\sigma,t) \models_{|\sigma|} Q\Big)\Big) \neq \mathit{false}$$

that can be reduced, according to the tri-valued logics semantics into the two following assertion:

$$\Big((\sigma,t) \models_{|\sigma|} P\Big) \neq \mathit{false} \vee \Big((\sigma,t) \models_{|\sigma|} Q\Big) \neq \mathit{false} \tag{A.135}$$

As assertion (A.48) is supposed to hold, we envision two cases: either the first disjunct holds or the second one.

- ***Case 1:*** $\Big((\sigma,t) \models_{|\sigma|} P\Big) \neq \mathit{false}$
  This case condition matches the antecedent of the second antecedent of (A.134), so that its consequent is supposed to hold. This leads to the following assertion:

  $$\mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \tag{A.136}$$

  On the other hand, the consequent of (A.134) can be reduced by expanding the bound function and according to the propositional logic semantics into the following:

  $$\begin{aligned} &\mathit{renaming}(\sigma) \models \mathit{bound}(P, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \\ &\vee \mathit{renaming}(\sigma) \models \mathit{bound}(Q, \mathit{weak}, |\sigma|, \mathit{renaming}, t) \end{aligned} \tag{A.137}$$

  That is trivially true because of (A.136).

- ***Case 2:*** $\Big((\sigma,t) \models_{|\sigma|} Q\Big) \neq \mathit{false}$
  The second case is identical to the first case with P being replaced by Q.

Q.E.D.

A.2.5.4 Soundness of Bounding Time for Assertions with Weakening

Lemma 20 *Under the assumption that the bounding function is sound for atomic assertions, it is also sound for assertions. The latter is formally defined below:*

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming, t)\\
&\quad 1 \leq t \leq |\sigma|\\
&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} P\Big) \neq false\\
&\quad \rightarrow renaming(\sigma) \models bound(P, strong, |\sigma|, renaming, t) \quad \text{(A.138)}
\end{aligned}
$$

Proof:

The proof is made of six cases matching the grammar on Assertions.

- ***Case 1:*** *P is of the pattern $\bullet Q$*
  The proof goal reduces to

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming, t)\\
&\quad (\forall t')\Big(1 \leq t' \leq |\sigma| \wedge \Big((\sigma, t') \models_{|\sigma|} Q\Big) \neq false\\
&\qquad\qquad \rightarrow renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t')\Big)\\
&\quad \wedge 1 \leq t \leq |\sigma|\\
&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} \bullet Q\Big) \neq false\\
&\quad \rightarrow renaming(\sigma) \models bound(\bullet Q, strong, |\sigma|, renaming, t) \quad \text{(A.139)}
\end{aligned}
$$

  This is proven by Lemma 15.

- ***Case 2:*** *P is of the pattern $\circ Q$*
  The proof goal reduces to

$$
\begin{aligned}
&(\forall\ \sigma, P : Assertion, renaming, t)\\
&\quad (\forall t')\Big(1 \leq t' \leq |\sigma| \wedge \Big((\sigma, t') \models_{|\sigma|} Q\Big) \neq false\\
&\qquad\qquad \rightarrow renaming(\sigma) \models bound(Q, strong, |\sigma|, renaming, t')\Big)\\
&\quad \wedge 1 \leq t \leq |\sigma|\\
&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} \circ Q\Big) \neq false\\
&\quad \rightarrow renaming(\sigma) \models bound(\circ Q, weak, |\sigma|, renaming, t) \quad \text{(A.140)}
\end{aligned}
$$

This is proven by Lemma 16.

- ***Case 3:*** *P is of the pattern $\Diamond Q$*
  The proof goal reduces to:

$$\begin{aligned}&(\forall\ \sigma, P : Assertion, renaming, t)\\&\quad \wedge 1 \leq t \leq |\sigma|\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} \circ Q\Big) \neq false\\&\quad \rightarrow renaming(\sigma) \models bound(\circ Q, weak, |\sigma|, renaming, t)\end{aligned} \tag{A.141}$$

  This is proven by Lemma 17

- ***Case 4:*** *P is of the pattern $R \wedge S$*
  The proof goal reduces to:

$$\begin{aligned}&(\forall\ \sigma, R : Assertion, S : Assertion, renaming, t)\\&\quad 1 \leq t \leq |\sigma|\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} R\Big) \neq false\\&\qquad \rightarrow renaming(\sigma) \models bound(R, weak, |\sigma|, renaming, t)\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} S\Big) \neq false\\&\qquad \rightarrow renaming(\sigma) \models bound(S, weak, |\sigma|, renaming, t)\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} R \wedge S\Big) \neq false\\&\quad \rightarrow renaming(\sigma) \models bound(R \wedge S, weak, |\sigma|, renaming, t)\end{aligned} \tag{A.142}$$

  This is proven by Lemma 18

- ***Case 5:*** *P is of the pattern $R \vee S$*
  The proof goal reduces to:

$$\begin{aligned}&(\forall\ \sigma, R : Assertion, Q : Assertion, renaming, t)\\&\quad 1 \leq t \leq |\sigma|\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} R\Big) \neq false\\&\qquad \rightarrow renaming(\sigma) \models bound(R, weak, |\sigma|, renaming, t)\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} S\Big) \neq false\\&\qquad \rightarrow renaming(\sigma) \models bound(S, weak, |\sigma|, renaming, t)\\&\quad \wedge\Big((\sigma, t) \models_{|\sigma|} R \vee S\Big) \neq false\\&\quad \rightarrow renaming(\sigma) \models bound(r \vee S, weak, |\sigma|, renaming, t)\end{aligned} \tag{A.143}$$

  This is proven by Lemma 19

- ***Case 6:*** *P is an atomic assertion*
  This case has already been discussed at the end of Section A.2.4.4

Q.E.D.

A.3 Bounding Integers in CONCHITA

This section describes how to propositionalize integers without strengtening. The basic technique for propositionalizing integers consists in bounding them to a finite range $\{0, 1, ..., n\}$. This technique inserts an additionnal constraint that is "every integer is lower than n". This is not desirable while propositionalizing integers in the knowledge of the attacker.

This section proposes another propositionalization of integers based on abstract interpretation techique. The basic idea is that integers are bounded to a finite number of value that contains an abstraction for all the other values, that is, they are propositionalized into the finite range: $\{0, 1, ..., n, \text{more}\}$ where "more" represents all the other values for integers. This abstraction imposes that all operators on integers must be translated to as to handle this abstract value. We focus on the following operators: $<, \leq, =, +, *$. Other operators are unsupported.

Assertions must be translated so as to replace the classical operators with the abstracted version of the operators. In order to be compatible with the overall approach, we provide two translations for expressions: one with strenghtening (that evaluates more to false) and one with weakening (that evaluates more to true). We want the abstraction to enforce the following properties:

$$
\begin{aligned}
&(\forall I_a, E)\Big(val_a(I_a, Abstract(E, strong, n)) \\
&\qquad \rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c, E)\Big)\Big) \\
&(\forall I_c, E)\Big(val_c(I_c, E) \\
&\qquad \rightarrow val_a(abs_n(I_c), Abstract(E, weak, n))\Big)
\end{aligned}
$$

Where:

- I_a represents an interpretation over abstract integer domains $\{0, 1, ..., n, \text{more}\}$
- I_c represents an interpretation over concrete integer domains $\{0, 1, ..., n, ...\}$
- E represents an assertion built with constants, variables and the connectives $\{<, \leq, = +, *\}$. The grammar of assertions is given below.
- val_a represents the function that maps an abstracted expresion and an interpretation over abstract domains onto a boolean value representing the evaluation of the expression on the interpretation.
- val_c represents the function that maps a concrete expression and an interpretation over concrete domains onto a boolean value representing the evaluation of the expression on the interpretation.
- $Abstract(E, strong, n)$ represents the function that abstracts an expression on integer variables into an abstracted expression on abstracted integer variables. This translation is performed with strenghtening.
- $Abstract(E, strong, n)$ represents the function that abstracts an expression on integer variables into an abstracted expression on abstracted integer variables. This translation is performed with weakening.
- abs_n represents a function that maps an interpretation over concrete integer domains into an abstraction over abstracted integer domains. The abstraction is parametrized with n, the upper integer value that belongs to the the abstract integer domain.

The abstraction process is presented and demonstrated bottom-up, starting from the $+$ and $*$ operators up to the $<$, $\leq$ and $=$ operators. The grammar for the assertions on integers is as follows:

```
Assertion ::=
[] Expression < Expression
[] Expression =< Expression
[] Expression = Expression

Expression ::=
[] Constant
[] Variable
[] Expression + Expression
[] Expression * Expression
```

The remaining of this chapter is organized as follows:
Section A.3.1 presents the abstraction of interpretations
Section A.3.2 presents the abstraction of expressions on integers
Section A.3.3 presents the abstraction of assertions on integers with strengthening
Section A.3.4 presents the abstraction of assertions on integers with weakening

A.3.1 Defining the Abstraction Functions

We can define the abstraction function for integer values by surcharging the abs_n function that was defined for abstracting interpretations:

$$abs_n(v) =_{def} \textbf{if } v > n \textbf{ then } \text{more}\textbf{else } v \textbf{ fi}$$

Interpretations and abstract interpretations are functions from variables to integer and abstracted integer values, respectively.

Concrete interpretations over integers are abstracted by the abs_n function as follows:

$$abs_n(I_c) =_{def} \lambda x.abs_n(I_c(x))$$

A.3.2 Abstracting Expressions

This section describes how expressions are abstracted on the basis of the abstration of integers. This section defines the val_a and the *Abstract* function on expressions. Moreover, we will prove the soundness and coverness of our abstraction process. These demonstrations are presented in Section A.3.2.5

and Section A.3.2.6 and build on lemmas presented along with the abstraction of the various grammar rules of expressions.

Note that the strong/weak argument of the *Abstract* function is irrelevant while abstracting expressions because expressions evaluate to integer value and not bolean, therefore it is denoted "_" in this section.

A.3.2.1 Abstracting Constants

A constant is abstracted by applying the abs_n function on its value. Therefore, the *Abstract* function is defined on constants as follows:

$$Abstract(C, _, n) =_{def} abs_n(C) \tag{A.144}$$

And the val_a function is defined on abstracted constant as follows:

$$val_a(I_a, C_a) =_{def} C_a \tag{A.145}$$

We can prove the soundness of this definition:

Lemma 21 *Soundness of abstraction of constants*

The abstraction of constants is sound, ie, it enforces the following property:

$$(\forall I_c) val_a(abs_n(I_c), Abstract(C, _, n)) = abs_n(val_c(I_c, C)) \tag{A.146}$$

Proof:

This is proven by simplifying (A.146).
Expanding the Abstract function and the val_a function with their definition in the left hand side yields:
$abs_n(C)$
Expanding the val_c function on its argument in the right hand side yields:
$abs_n(C)$
The reduction of the two sides are equal.

Q.E.D.

A.3.2.2 Abstracting Variables

A variable is abstracted by itself. Basically, its domain changes and the result of evaluating the value of a variable over an abstract interpretation

yields an abstract value. Therefore, the *Abstract* function is defined on variables as follows:

$$Abstract(Var, _, n) =_{def} Var \tag{A.147}$$

And the val_a function is defined on variables as follows:

$$val_a(I_a, Var) =_{def} I_a(Var) \tag{A.148}$$

Lemma 22 *Soundness of abstraction of variables*

The abstraction of variables is sound, ie, it enforces the following property:

$$(\forall I_c) val_a(abs_n(I_c), Abstract(Var, _, n)) = abs_n(val_c(I_c, Var)) \tag{A.149}$$

Proof:

This is proven by simplifying (A.149).
Expanding the Abstract function with its definition in the left hand side of (A.149) yields:
$val_a(abs_n(I_c), Var)$
This is further simplified by expanding the abs_n function. This yields:
$val_a(\lambda x.abs_n(I_c(x)), Var)$
And the val_a function is unfolded, this yields:
$abs_n(I_c(Var))$
Expanding the val_c function in the right hand side of (A.149) yields:
$abs_n(I_c(Var))$
The reduction of the two sides are equal.

Q.E.D.

A.3.2.3 Abstracting Sums

The + operator is abstracted into the $+_{a,n}$ operator. Therefore, the *Abstract* function is defined on sums as follows:

$$\begin{aligned} &Abstract(\text{"}E + F\text{"}, _, n) =_{def} \\ &\quad Abstract(\text{"}E\text{"}, _, n) +_{a,n} Abstract(\text{"}F\text{"}, _, n) \end{aligned} \tag{A.150}$$

The semantics of $+_{a,n}$ is defined by the val_a function as follows:

$$val_a(I_a, \text{"}E +_{a,n} F\text{"}) =_{def}$$

$$\begin{array}{l}\textbf{if } val_a(I_a,"E") = \text{more} \\ \quad \vee val_a(I_a,"F") = \text{more} \\ \quad \vee val_a(I_a,"E") + val_a(I_a,"F") > n \\ \textbf{then} \\ \quad \text{more} \\ \textbf{else} \\ \quad val_a(I_a,"E") + val_a(I_a,"F") \\ \textbf{fi}\end{array} \tag{A.151}$$

Lemma 23 *Soundness of abstraction of sums*

Let be E and F, expressions such that

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E", _, n)) = abs_n(val_c(I_c,"E")) \tag{A.152}$$
$$(\forall I_c) val_a(abs_n(I_c), Abstract("F", _, n)) = abs_n(val_c(I_c,"F")) \tag{A.153}$$

The abstraction of the + opererator defined in (A.150) and (A.151) enforces property (A.154):

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E + F", _, n)) = abs_n(val_c(I_c,"E + F")) \tag{A.154}$$

Proof:

The proof proceeds by a series of cases on the properties of the values of E and F under interpretation I_c. For each case, we reduce both sides of of (A.154) and show that they turn out to be equal. The values of E and F under interpretaton I_c are denoted V_E and V_F for short.

Before detailing the different cases, we perform some preliminary simplifications on (A.154). Using the definition of Abstract, we can reduce the left hand side of (A.154) into:

$$val_a(abs_n(I_c), Abstract("E", _, n) +_{a,n} Abstract("F", _, n)) \tag{A.155}$$

By expanding the val_a function, (A.155) is reduced into:

$$
\begin{array}{l}
\textbf{\textit{if}} \\
\quad val_a(abs_n(I_c), Abstract(\text{"}E\text{"}, _, n)) = more \\
\quad \vee val_a(abs_n(I_c), Abstract(\text{"}F\text{"}, _, n)) = more \\
\quad \vee val_a(abs_n(I_c), Abstract(\text{"}E\text{"}, _, n)) \\
\qquad\quad + val_a(abs_n(I_c), Abstract(\text{"}F\text{"}, _, n)) > n \\
\textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad val_a(abs_n(I_c), Abstract(\text{"}E\text{"}, _, n)) \\
\qquad + val_a(abs_n(I_c), Abstract(\text{"}F\text{"}, _, n)) \\
\textbf{\textit{fi}}
\end{array}
$$

That can be simplified with the hypothesis (A.152) and (A.153) and using the V_E and V_F abbreviations into:

$$
\begin{array}{l}
\textbf{\textit{if}} \\
\quad abs_n(V_E) = more \\
\quad \vee abs_n(V_F) = more \\
\quad \vee abs_n(V_E) + abs_n(V_F) > n \\
\textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad abs_n(V_E) + abs_n(V_F) \\
\textbf{\textit{fi}}
\end{array}
\tag{A.156}
$$

We also reduce the right hand side of (A.154) by expanding the concrete sum operator. This yields the following:

$$abs_n(V_E + V_F) \tag{A.157}$$

We now simplify (A.156) and (A.157) further in a series of cases in order to show that they are always equal.

- ***Case 1:*** $V_E + V_F <= n$
  In this case, we can reduce further (A.157) into $V_E + V_F$.
  As both V_E and V_F are lower than n, we have that:

$$
\begin{aligned}
abs_n(V_E) &= V_E \\
abs_n(V_F) &= V_F
\end{aligned}
$$

and (A.156) is reduced to:

$$
\begin{array}{l}
\textbf{\textit{if}} \\
\quad V_E = more \\
\quad \vee V_F = more \\
\quad \vee V_E + V_F > n \\
\textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad V_E + V_F \\
\textbf{\textit{fi}}
\end{array}
$$

According to the case condition, the condition evaluates to false. Therefore, this formula reduces to: $V_E + V_F$*, that is equal to the reduction of (A.157)*

- ***Case 2:*** $V_E + V_F > n$
  In this case, we can reduce further (A.157) into: more. On the other hand, (A.156) is reduced further with a nested series on cases on V_E *and* V_F*.*

  - ***Case 2.1:*** $V_E > n$ *and* $V_F > n$
    In this case (A.156) is reduced to:

$$
\begin{array}{l}
\textbf{\textit{if}} \\
\quad more = more \\
\quad \vee more = more \\
\quad \vee more + more > n \\
\textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad V_E + V_F \\
\textbf{\textit{fi}}
\end{array}
$$

    That obviously reduces to more, that is equal to the reduction of (A.157).
  - ***Case 2.2:*** $V_E > n$ *and* $V_F \leq n$
    In this case (A.156) is reduced to:

$$
\begin{array}{l}
\textbf{\textit{if}} \\
\quad more = more \\
\quad \vee V_F = more \\
\quad \vee more + V_F > n \\
\textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad more + V_F \\
\textbf{\textit{fi}}
\end{array}
$$

That obviously reduces to more, that is equal to the reduction of (A.157).

- ***Case 2.3:*** $V_E \leq n$ *and* $V_F > n$
  This case is symmetric to ***case 2.2*** *and leads to the same conclusion.*
- ***Case 2.4:*** $V_E \leq n$ *and* $V_F \leq n$
  In this case, (A.156) is reduced to:

$$
\begin{array}{l}
\textbf{\textit{if}} \\
\quad V_E = more \\
\quad \vee V_F = more \\
\quad \vee V_E + V_F > n \\
\textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad V_E + V_F \\
\textbf{\textit{fi}}
\end{array}
$$

The third disjunct of the condition evaluates to true because of the condition of ***case 2****. Therefore, this expression reduces to more, that is equal to the reduction of (A.157).*

Q.E.D.

A.3.2.4 Abstracting Products

The $*$ operator is abstracted into the $*_{a,n}$ operator. Therefore, the *Abstract* function is defined on products as follows:

$$
\begin{aligned}
&Abstract(\text{"}E * F\text{"}, _, n) \\
&\qquad =_{def} Abstract(\text{"}E\text{"}, _, n) *_{a,n} Abstract(\text{"}F\text{"}, _, n)
\end{aligned}
\tag{A.158}
$$

The semantics of $*_{a,n}$ is defined by the val_a function as follows:

$$
val_a(I_a, \text{"}E *_{a,n} F\text{"}) =_{def}
$$

$$
\begin{array}{l}
\textbf{if} \quad val_a(I_a, "E") = 0 \vee val_a(I_a, "F") = 0 \textbf{ then} \\
\quad 0 \\
\textbf{elseif } val_a(I_a, "E") = \text{more} \\
\quad \vee val_a(I_a, "F") = \text{more} \\
\quad \vee val_a(I_a, "E") * val_a(I_a, "F") > n \\
\textbf{ then} \\
\quad \text{more} \\
\textbf{else} \\
\quad val_a(I_a, "E") * val_a(I_a, "F") \\
\textbf{ fi}
\end{array}
\tag{A.159}
$$

Lemma 24 *Soundness of abstraction of product*

Let be E and F, expressions such that

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E", _, n)) = abs_n(val_c(I_c, "E")) \tag{A.160}$$
$$(\forall I_c) val_a(abs_n(I_c), Abstract("F", _, n)) = abs_n(val_c(I_c, "F")) \tag{A.161}$$

The abstraction of the $*$ opererator defined in (A.159) and (A.158) enforces property (A.162):

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E * F", _, n)) = abs_n(val_c(I_c, "E * F")) \tag{A.162}$$

Proof:

The proof proceeds by a series of cases on the properties of the values of E and F under interpretation I_c. For each case, we reduce both sides of of (A.162) and show that they turn out to be equal. The values of E and F under interpretaton I_c are denoted V_E and V_F for short.

Before detailing the different cases, we perform some preliminary simplifications on (A.162). Using the definition of Abstract, we can reduce the left hand side of (A.162) into:

$$val_a(abs_n(I_c), Abstract("E", _, n) *_{a,n} Abstract("F", _, n)) \tag{A.163}$$

By expanding the val_a function, (A.163) is reduced into:

$$
\begin{array}{l}
\textbf{\textit{if}} \quad val_a(abs_n(I_c),"E") == 0 \vee val_a(abs_n(I_c),"F") = 0 \ \textbf{\textit{then}} \\
\quad O \\
\textbf{\textit{elseif}}\ val_a(abs_n(I_c),"E") = more \\
\quad \vee val_a(abs_n(I_c),"F") = more \\
\quad \vee val_a(abs_n(I_c),"E") * val_a(abs_n(I_c),"F") > n \ \textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad val_a(abs_n(I_c),"E") * val_a(abs_n(I_c),"F") \\
\textbf{\textit{fi}}
\end{array}
$$

That can be simplified with the hypothesis (A.160) and (A.161) and using the V_E and V_F abbreviations into:

$$
\begin{array}{l}
\textbf{\textit{if}}\ V_E = 0 \vee V_F = 0 \ \textbf{\textit{then}} \\
\quad 0 \\
\textbf{\textit{elseif}}\ V_E = more \\
\quad \vee V_F = more \\
\quad \vee V_E * V_F > n \ \textbf{\textit{then}} \\
\quad more \\
\textbf{\textit{else}} \\
\quad V_E * V_F \\
\textbf{\textit{fi}}
\end{array}
\tag{A.164}
$$

We also reduce the right hand side of (A.162) by expanding the concrete product operator. This yields the following:

$$abs_n(V_E * V_F) \tag{A.165}$$

We now simplify the two equations further in a series of cases and show that they are always equal.

- ***Case 1:*** $V_E = 0 \vee V_F = 0$
  In this case, (A.165) evaluates to 0.
  Moreover, (A.164) reduces to 0 *because its first condition evaluates to true. Therefore, it is equal to the reduction of (A.165).*

- ***Case 2:*** $V_E \neq 0 \wedge V_F \neq 0 \wedge V_E * V_F <= n$
  In this case, we can reduce further (A.165) into $V_E * V_F$.

  As V_E *and* V_F *are positive non null integer values, they are both lower than* n*, so that we have:*

$$
\begin{aligned}
abs_n(V_E) &= V_E \\
abs_n(V_F) &= V_F
\end{aligned}
$$

and (A.164) reduces into:

$$\begin{array}{l}\textbf{\textit{if}}\ V_E = more \\ \quad \vee V_F = more \\ \quad \vee V_E * V_F > n\ \textbf{\textit{then}} \\ \quad more \\ \textbf{\textit{else}} \\ \quad V_E * V_F \\ \textbf{\textit{fi}}\end{array}$$

According to the case condition, the condition evaluates to false. Therefore, this formula is reduced to: $V_E * V_F$*, that is equal to the reduction of (A.165).*

- ***Case 3:*** $V_E \neq 0 \wedge V_F \neq 0 \wedge V_E * V_F > n$
  In this case, we can reduce further (A.165) into: more. (A.164) is reduced further with a nested series on cases on V_E *and* V_F*.*

  - ***Case 3.1:*** $V_E > n$ *and* $V_F > n$
    In this case (A.164) is reduced into:

$$\begin{array}{l}\textbf{\textit{if}}\ more = more \\ \quad \vee more = more \\ \quad \vee more * more > n\ \textbf{\textit{then}} \\ \quad more \\ \textbf{\textit{else}} \\ \quad more * more \\ \textbf{\textit{fi}}\end{array}$$

    That obviously reduces to more, that is equal to the reduction of (A.165).

  - ***Case 3.2:*** $V_E > n$ *and* $V_F \leq n$
    In this case (A.164) is reduced to:

$$\begin{array}{l}\textbf{\textit{if}} \\ \quad more = more \\ \quad \vee V_F = more \\ \quad \vee more * V_F > n \\ \textbf{\textit{then}} \\ \quad more \\ \textbf{\textit{else}} \\ \quad more * V_F \\ \textbf{\textit{fi}}\end{array}$$

    That reduces to more because the first disjunct of the condition evaluates to true. It is therefore equal to the reduction of (A.165).

- ***Case 3.3:*** $V_E \leq n$ *and* $V_F > n$
  This case is symmetric to ***Case 3.2*** *and leads to the same conclusion.*
- ***Case 3.4:*** $V_E \leq n$ *and* $V_F \leq n$
  In this case (A.164) is reduced to:

$$\begin{array}{l} \textbf{\textit{if}} \\ \quad V_E = V_F \\ \quad \vee V_F = V_F \\ \quad \vee V_E * V_F > n \\ \textbf{\textit{then}} \\ \quad more \\ \textbf{\textit{else}} \\ \quad V_E * V_F \\ \textbf{\textit{fi}} \end{array}$$

  The third disjunct of the condition evaluates to true because of the condition of ***case 3****. Therefore, this expression reduces to more, that is equal to the reduction of (A.165).*

Q.E.D.

A.3.2.5 Soundness of the Abstraction of Expressions

Theorem 5 *Soundness of abstraction of expressions*

The abstraction for expression is sound; i,e: the following assertion holds:

$$(\forall E, I_c) val_a(abs_n(I_c), Abstract(E, _, n)) = abs_n(val_c(I_c, E)) \quad \text{(A.166)}$$

Proof:

This demonstration is done by induction over the grammar of expressions. The base cases are the Constant and the Variable expressions and the inductive cases are the + and the ∗-based expressions.

- ***Base Case of constants***

$$(\forall I_c) val_a(abs_n(I_c), Abstract(C, _, n)) = abs_n(val_c(I_c, C))$$

This case is demonstrated by Lemma 21.

- ***Base case of variables***

$$(\forall I_c) val_a(abs_n(I_c), Abstract(Var, _, n)) = abs_n(val_c(I_c, Var))$$

This case is demonstrated by Lemma 22.

- ***Induction case of sums***

Let be E and F, expressions such that (induction hypothesis):

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E", _, n)) = abs_n(val_c(I_c, "E"))$$
$$(\forall I_c) val_a(abs_n(I_c), Abstract("F", _, n)) = abs_n(val_c(I_c, "F"))$$

Goal of the proof:

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E + F", _, n)) = abs_n(val_c(I_c, "E + F"))$$

This case is demonstrated by Lemma 23.

- ***Induction case of products***

Let be E and F, expressions such that (induction hypothesis):

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E", _, n)) = abs_n(val_c(I_c, "E"))$$
$$(\forall I_c) val_a(abs_n(I_c), Abstract("F", _, n)) = abs_n(val_c(I_c, "F"))$$

Proof goal:

$$(\forall I_c) val_a(abs_n(I_c), Abstract("E * F", _, n)) = abs_n(val_c(I_c, "E * F"))$$

This case is demonstrated by Lemma 24.

Q.E.D.

A.3.2.6 Coverness of the Abstraction of Expressions

Lemma 25 *The interpretation I_c constructed as follows:*

$$I_c(var) = \textbf{\textit{if}}\ I_a(var) = more\ \textbf{\textit{then}}\ n + 1 \textbf{\textit{else}}\ I_a(var)\ \textbf{\textit{fi}}$$

is a possible concretization of I_a, ie: $I_a = abs_n(I_c)$.

Proof:

This is proven by showing that for all variable the abstraction of the I_c constructed above always yields the same value as the original I_a:

$$
\begin{aligned}
(\forall Var)\Big(I_a(Var) \quad = \quad & abs_n(\lambda var.(\\
& \quad \textbf{\textit{if}}\ I_a(var) = more\ \textbf{\textit{then}}\ n+1 \textbf{\textit{else}}\ I_a(var)\ \textbf{\textit{fi}} \\
&))(Var)\Big)
\end{aligned}
$$

By expanding the abs_n function on interpretation, this formula is reduced to:

$$
\begin{aligned}
(\forall Var)\Big(I_a(Var) \quad = \quad & \lambda var.abs_n(\\
& \quad \textbf{\textit{if}}\ I_a(var) = more\ \textbf{\textit{then}}\ n+1 \textbf{\textit{else}}\ I_a(var)\ \textbf{\textit{fi}} \\
&)(Var)\Big)
\end{aligned}
$$

By expanding the abs_n function abain, but on values, the formula is reduced to:

$$
\begin{array}{l}
(\forall Var) I_a(Var) = \\
\quad \textbf{\textit{if}} \\
\quad\quad (\textbf{\textit{if}}\ I_a(Var) = more\ \textbf{\textit{then}} \\
\quad\quad\quad n+1 \\
\quad\quad \textbf{\textit{else}} \\
\quad\quad\quad I_a(Var) \\
\quad\quad \textbf{\textit{fi}}) > n\ \textbf{\textit{then}} \\
\quad\quad more \\
\quad \textbf{\textit{else}} \\
\quad\quad \textbf{\textit{if}}\ I_a(Var) = more\ \textbf{\textit{then}} \\
\quad\quad\quad n+1 \\
\quad\quad \textbf{\textit{else}} \\
\quad\quad\quad I_a(Var) \\
\quad\quad \textbf{\textit{fi}} \\
\quad \textbf{\textit{fi}}
\end{array}
\tag{A.167}
$$

Let be a certain variable Var, we reduce the right-hand side of (A.167) on different cases on $I_a(Var)$ and show that it always turn out to be equal to the left-hand side member of (A.167).

- ***Case 1:*** $I_a(Var) = more$
  In this case, the nested conditionals conditions reduce to true. Therefore, the right-hand side of (A.167) is reduced as follows:

$$\begin{array}{l} \textbf{\textit{if}}\ n+1 > n\ \textbf{\textit{then}} \\ \quad more \\ \textbf{\textit{else}} \\ \quad ... \\ \textbf{\textit{fi}} \end{array}$$

  That trivially reduces to more. Due to the condition of ***case 1****, the left hand side of (A.167) also reduces to more.*

- ***Case 2:*** $I_a(Var) \neq more$
  In this case, the nested conditionals conditions of the right-hand side member of (A.167) reduce to false. Therefore, the right-hand side member of (A.167) is reduced as follows:

$$\begin{array}{l} \textbf{\textit{if}}\ I_a(Var) > n\ \textbf{\textit{then}} \\ \quad more \\ \textbf{\textit{else}} \\ \quad I_a(Var) \\ \textbf{\textit{fi}} \end{array}$$

  The condition evaluates to false therefore, the right hand side evaluates to $I_a(Var)$, that is equivalent to the left hand side of (A.167).

Q.E.D.

Theorem 6 *Coverness of abstraction of expressions*

The abstraction for expression covers only existing cases; i,e: the following assertion holds:

$$\begin{array}{l} (\forall\ E, I_a)(\exists I_c) \\ \qquad \Big(val_a(I_a, Abstract(E, _, n)) = abs_n(val_c(I_c, E)) \\ \qquad\ \wedge I_a = abs_n(I_c)\Big) \end{array} \tag{A.168}$$

Proof:

We use the I_c constructed in Lemma 25. For this I_c, Lemma 25 demonstrates independently of E that it enforces the second conjunct of the theorem. Therefore, the first conjunct is demonstrated by Theorem 5.

Q.E.D.

A.3.3 Abstracting Assertions on Integers with Strengtening

This section describes how assertions on integers are abstracted on the basis of the abstration of integers. This section defines the val_a and the *Abstract* function on assertions while abstracting with strengtening. As in the previous section, we prove the soundness of our abstraction process. This demonstraction is presented in Section A.3.3.4 and builds on lemmas presented along with the abstraction of the various grammar rules of expressions.

A.3.3.1 Abstracting Equality

The equality operator is abstracted with strengtening into the $=_{a,n,strong}$ operator. Therefore, the *Abstract* function is defined on equality as follows:

$$\begin{aligned} &Abstract(\text{"}E = F\text{"}, strong, n) =_{def} \\ &\quad Abstract(\text{"}E\text{"}, strong, n) =_{a,n,strong} Abstract(\text{"}F\text{"}, strong, n) \end{aligned} \tag{A.169}$$

The semantics of $=_{a,n,strong}$ is defined by the val_a function as follows:

$$\begin{aligned} &val_a(I_a, \text{"}E =_{a,n,strong} F\text{"}) =_{def} \\ &\quad \textbf{if} \\ &\quad\quad val_a(I_a, \text{"}E\text{"}) = \text{more} \\ &\quad\quad \vee val_a(I_a, \text{"}F\text{"}) = \text{more} \\ &\quad \textbf{then} \\ &\quad\quad \text{false} \\ &\quad \textbf{else} \\ &\quad\quad val_a(I_a, \text{"}E\text{"}) = val_a(I_a, \text{"}F\text{"}) \\ &\quad \textbf{fi} \end{aligned} \tag{A.170}$$

We can prove the soundness of this definition:

Lemma 26 *Soundness of strong abstraction of equality*

$$\begin{aligned} &(\forall I_a)\Big(val_a(I_a, Abstract(\text{"}E = F\text{"}, strong, n)) \\ &\quad \rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c, \text{"}E = F\text{"})\Big)\Big) \end{aligned} \tag{A.171}$$

Proof:

We will demonstrate that the right hand side is true for any given I_a, assuming that the left hand side is true. In the right hand side, we use the

I_c constructed in Lemma 25. For this I_c, Lemma 25 demonstrates that the first conjunct of the right hand side is true. Therefore, we only have to prove that $val_c(I_c, "E = F")$ evaluates to true.

The left hand side of (A.171) can be reduced by expanding the Abstract function and the val_a function as follows:

if
$val_a(I_a, Abstract("E", _, n)) = more$
$\vee val_a(I_a, Abstract("F", _, n)) = more$
then
false
else
$val_a(I_a, Abstract("E", _, n)) = val_a(I_a, Abstract("F", _, n))$
fi

As we made the hypothesis that this formula evaluates to true, the condition evaluates to false and the second condition evaluates to true, leading to the following assertions:

$$val_a(I_a, Abstract("E", _, n)) \neq more \quad (A.172)$$
$$val_a(I_a, Abstract("F", _, n)) \neq more \quad (A.173)$$
$$val_a(I_a, Abstract("E", _, n)) = val_a(I_a, Abstract("F", _, n)) \quad (A.174)$$

By applying Theorem 5 on (A.172), (A.173) and (A.174), we can simplify them as follows:

$$abs_n(val_c(I_c, "E")) \neq more \quad (A.175)$$
$$abs_n(val_c(I_c, "F")) \neq more \quad (A.176)$$
$$abs_n(val_c(I_c, "E")) = abs_n(val_c(I_c, "F")) \quad (A.177)$$

$val_c(I_c, "E")$ and $val_c(I_c, "F")$ evaluate to integers that are lower or equal to n because of (A.175) and (A.176), therefore, their abstraction is identical to their concrete values, so that (A.177) reduces to $val_c(I_c, "E = F")$ where val_c has been expanded:

$$val_c(I_c, "E") = val_c(I_c, "F")$$

Q.E.D.

A.3.3.2 Abstracting "Lower than"

The equality operator is abstracted with strengtening into the $<_{a,n,strong}$ operator. Therefore, the *Abstract* function is defined on $<$ as follows:

$$\begin{aligned} &Abstract(\text{"}E < F\text{"}, strong, n) =_{def} \\ &\quad Abstract(\text{"}E\text{"}, strong, n) <_{a,n,strong} Abstract(\text{"}F\text{"}, strong, n) \end{aligned} \tag{A.178}$$

The semantics of $<_{a,n,strong}$ is defined by the val_a function as follows:

$$\begin{aligned} &val_a(I_a, \text{"}E <_{a,n,strong} F\text{"}) =_{def} \\ &\qquad \textbf{if } val_a(I_a, \text{"}E\text{"}) = \text{more } \textbf{then} \\ &\qquad\quad \text{false} \\ &\qquad \textbf{elseif } val_a(I_a, \text{"}F\text{"}) = \text{more } \textbf{then} \\ &\qquad\quad \text{true} \\ &\qquad \textbf{else} \\ &\qquad\quad val_a(I_a, \text{"}E\text{"}) < val_a(I_a, \text{"}F\text{"}) \\ &\qquad \textbf{fi} \end{aligned} \tag{A.179}$$

We can prove the soundness of this definition:

Lemma 27 *Soundness of strong abstraction of "<"*

$$\begin{aligned} (\forall I_a)\Big(&val_a(I_a, Abstract(\text{"}E < F\text{"}, strong, n)) \\ &\rightarrow (\exists I_c)(I_a = abs_n(I_c) \wedge val_c(I_c, \text{"}E < F\text{"}))\Big) \end{aligned} \tag{A.180}$$

Proof:

We will demonstrate that the right hand side is true for any given I_a, assuming that the left hand side is true. In the right hand side, we use the I_c constructed in Lemma 25. For this I_c, Lemma 25 demonstrates that the first conjunct of the right hand side is true. Therefore, we only have to prove that $val_c(I_c, \text{"}E < F\text{"})$ evaluates to true.

The left hand side can be reduced by expanding the Abstract function

and the val_a function as follows:

$$\begin{array}{l}\textbf{\textit{if}}\ val_a(I_a, Abstract(\text{"}E\text{"}, _, n)) = more\ \textbf{\textit{then}}\\ \quad false\\ \textbf{\textit{elseif}}\ val_a(I_a, Abstract(\text{"}F\text{"}, _, n)) = more\ \textbf{\textit{then}}\\ \quad true\\ \textbf{\textit{else}}\\ \quad val_a(I_a, Abstract(\text{"}E\text{"}, _, n)) < val_a(I_a, Abstract(\text{"}F\text{"}, _, n))\\ \textbf{\textit{fi}}\end{array} \tag{A.181}$$

As we made the hypothesis that this formula evaluates to true, the first condition evaluates to false. We can therefore use the following assertion:

$$val_a(I_a, Abstract(\text{"}E\text{"}, _, n)) \neq more$$

This formula can be reduced by Theorem 5 into:

$$abs_n(val_c(I_c, E)) \neq more$$

This equation enforces that $val_c(I_c, E)$ evaluates to an integer that is lower or equal to n:

$$val_c(I_c, \text{"}E\text{"}) \leq n \tag{A.182}$$

On the other hand, (A.181) reduces to:

$$\begin{array}{l}\textbf{\textit{if}}\ val_a(I_a, Abstract(\text{"}F\text{"}, _, n)) = more\ \textbf{\textit{then}}\\ \quad true\\ \textbf{\textit{else}}\\ \quad val_a(I_a, Abstract(\text{"}E\text{"}, _, n)) < val_a(I_a, Abstract(\text{"}F\text{"}, _, n))\\ \textbf{\textit{fi}}\end{array} \tag{A.183}$$

The remaining of the demonstration proceeds by two cases that correspond to the condition above:

- ***Case 1:*** $val_a(I_a, Abstract(\text{"}F\text{"}, _, n)) = more$
  By applying Theorem 5, the case condition can be rewritten as follows:

  $$abs_n(val_c(I_c, \text{"}F\text{"})) = more$$

  That expresses that $val_c(I_c, \text{"}F\text{"})$ evaluates to an integer bigger than n:

  $$val_c(I_c, \text{"}F\text{"}) > n$$

By chaining this assertion with (A.182), we have that:

$$val_c(I_c, "E") \leq n < val_c(I_c, "F")$$

That enforces the remaining conjunct of the right of (A.180), that is: $val_c(I_c, "E < F")$ *where the* val_c *function has been expanded:*

$$val_c(I_c, "E") < val_c(I_c, "F")$$

- ***Case 2:*** $val_a(I_a, Abstract("F", _, n)) \neq more$
  Because of (A.183), the following assertion holds:

$$val_a(I_a, Abstract("E", _, n)) < val_a(I_a, Abstract("F", _, n)) \quad \text{(A.184)}$$

By applying Theorem 5 on both members of (A.184), we get:

$$abs_n(val_c(I_c, "E")) < abs_n(val_c(I_c, "F")) \quad \text{(A.185)}$$

By applying Theorem 5, the case condtion can be rewritten as follows:

$$abs_n(val_c(I_c, "F")) \neq more$$

That expresses that $val_c(I_c, "F")$ *evaluates to an integer lower or equal to* n*:*

$$val_c(I_c, "F") \leq n$$

Therefore, both members of (A.185) are different from more, so that their abstract value and their concrete values are identical. That enforces the remaining conjunct of the right of (A.180), that is: $val_c(I_c, "E < F")$ *where the* val_c *function has been expanded:*

$$val_c(I_c, "E") < val_c(I_c, "F")$$

Q.E.D.

A.3.3.3 Abstracting "Lower or Equal To"

The $\leq$ operator is abstracted with strengtening into the $\leq_{a,n,strong}$ operator. Therefore, the *Abstract* function is defined on $\leq$ as follows:

$$Abstract("E \leq F", strong, n) =_{def} Abstract("E", strong, n) \leq_{a,n,strong} Abstract("F", strong, n) \quad \text{(A.186)}$$

The semantics of $\leq_{a,n,strong}$ is defined by the val_a function as follows:

$$
\begin{aligned}
val_a(I_a, "E \leq_{a,n,strong} F") =_{def} \\
&\textbf{if } val_a(I_a, "E") = \text{more } \textbf{then} \\
&\quad \text{false} \\
&\textbf{elseif } val_a(I_a, "F") = \text{more } \textbf{then} \\
&\quad \text{true} \\
&\textbf{else} \\
&\quad val_a(I_a, "E") \leq val_a(I_a, "F") \\
&\textbf{fi}
\end{aligned}
\tag{A.187}
$$

We can prove the soundness of this definition:

Lemma 28 *Soundness of strong abstraction of "$\leq$"*

$$
\begin{aligned}
(\forall I_a)\Big(&val_a(I_a, Abstract("E \leq F", strong, n)) \\
&\rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c, "E \leq F")\Big)\Big)
\end{aligned}
\tag{A.188}
$$

Proof:

The proof very similar to the one of Lemma 27.

We will demonstrate that the right hand side is true for any given I_a, assuming that the left hand side is true. In the right hand side, we use the I_c constructed in Lemma 25. For this I_c, Lemma 25 demonstrates that the first conjunct of the right hand side is true. Therefore, we only have to prove that $val_c(I_c, "E \leq F")$ evaluates to true.

The left hand side can be reduced by expanding the Abstract function and the val_a function as follows:

$$
\begin{aligned}
&\textbf{\textit{if}}\ val_a(I_a, Abstract("E", _, n)) = more\ \textbf{\textit{then}} \\
&\quad false \\
&\textbf{\textit{elseif}}\ val_a(I_a, Abstract("F", _, n)) = more\ \textbf{\textit{then}} \\
&\quad true \\
&\textbf{\textit{else}} \\
&\quad val_a(I_a, Abstract("E", _, n)) \leq val_a(I_a, Abstract("F", _, n)) \\
&\textbf{\textit{fi}}
\end{aligned}
\tag{A.189}
$$

As we made the hypothesis that this formula evaluates to true, the first condition evaluates to false. We can therefore use the following assertion:

$$val_a(I_a, Abstract("E", _, n)) \neq more$$

This formula can be reduced by Theorem 5 into:

$$abs_n(val_c(I_c, E)) \neq more$$

This equation enforces that $val_c(I_c, E)$ evaluates to an integer that is lower or equal to n:

$$val_c(I_c, "E") \leq n \tag{A.190}$$

On the other hand, (A.189) reduces to:

$$\begin{array}{l} \textbf{\textit{if}}\ val_a(I_a, Abstract("F", _, n)) = more\ \textbf{\textit{then}} \\ \quad true \\ \textbf{\textit{else}} \\ \quad val_a(I_a, Abstract("E", _, n)) \leq val_a(I_a, Abstract("F", _, n)) \\ \textbf{\textit{fi}} \end{array} \tag{A.191}$$

The remaining of the demonstration proceeds by two cases that correspond to the condition of (A.191):

- ***Case 1:*** $val_a(I_a, Abstract("F", _, n)) = more$
  By applying Theorem 5, the case condtion can be rewritten as follows:

  $$abs_n(val_c(I_c, "F")) = more$$

  That expresses that $val_c(I_c, "F")$ evaluates to an integer bigger than n:

  $$val_c(I_c, "F") > n$$

  By chaining this assertion with (A.190), we have that:

  $$val_c(I_c, "E") \leq n < val_c(I_c, "F")$$

  That enforces the remaining conjunct of the right of (A.188), that is: $val_c(I_c, "E \leq F")$ where the val_c function has been expanded:

  $$val_c(I_c, "E") \leq val_c(I_c, "F")$$

- ***Case 2:*** $val_a(I_a, Abstract("F", _, n)) \neq more$
  Because of (A.191), the following assertion holds:

  $$val_a(I_a, Abstract("E", _, n)) \leq val_a(I_a, Abstract("F", _, n)) \tag{A.192}$$

By applying Theorem 5 on both members of (A.192), we get:

$$abs_n(val_c(I_c,"E")) \leq abs_n(val_c(I_c,"F")) \tag{A.193}$$

By applying Theorem 5, the case condition can be rewritten as follows:

$$abs_n(val_c(I_c,"F")) \neq more$$

That expresses that $val_c(I_c,"F")$ evaluates to an integer lower or equal to n:

$$val_c(I_c,"F") \leq n$$

Therefore, both members of (A.193) are different from more, so that their abstract value and their concrete values are identical. That enforces the remaining conjunct of the right of (A.188), that is: $val_c(I_c,"E \leq F")$ where the val_c function has been expanded:

$$val_c(I_c,"E") \leq val_c(I_c,"F")$$

Q.E.D.

A.3.3.4 Soundness of the Strong Abstraction of Assertions

Theorem 7 *Soundness of strong abstraction of assertions on integers*

$$(\forall I_a, E)\Big(val_a(I_a, Abstract(E, strong, n)) \\ \rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c, E)\Big)\Big) \tag{A.194}$$

Proof:

According to the grammar of assertions, we have to consider three cases:

- ***E is an equality***
  The demonstration goal is then:

$$(\forall I_a)\Big(val_a(I_a, Abstract("E = F", strong, n)) \\ \rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c,"E = F")\Big)\Big)$$

  This is proven in Lemma 26.

- ***E is a "lower than"***
  The demonstration goal is then:

$$(\forall I_a)\Big(val_a(I_a, Abstract("E < F", strong, n)) \\ \rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c, "E < F")\Big)\Big)$$

  This is proven in Lemma 27.

- ***E is a "lower or equal to"***
  The demonstration goal is then:

$$(\forall I_a)\Big(val_a(I_a, Abstract("E \leq F", strong, n)) \\ \rightarrow (\exists I_c)\Big(I_a = abs_n(I_c) \wedge val_c(I_c, "E \leq F")\Big)\Big)$$

  This is proven in Lemma 28.

Q.E.D.

A.3.4 Abstracting Assertions on Integers with Weakening

This section describes how assertions on integers are abstracted on the basis of the abstration of integers. This section defines the val_a and the *Abstract* function on assertions while abstracting with weakening. As in the previous section, we prove the soundness of our abstraction process. This demonstraction is presented in Section A.3.4.4 and builds on lemmas presented along with the abstraction of the various grammar rules of expressions.

A.3.4.1 Abstracting Equality

The equality operator is abstracted with weakening into the $=_{a,n,weak}$ operator. Therefore, the *Abstract* function is defined on equality as follows:

$$Abstract("E = F", weak, n) =_{def} \\ Abstract("E", weak, n) =_{a,n,weak} Abstract("F", weak, n) \quad \text{(A.195)}$$

The semantics of $=_{a,n,weak}$ is defined by the val_a function as follows:

$$val_a(I_a, "E =_{a,n,weak} F") =_{def}$$

$$\begin{array}{l}\textbf{if}\\ \quad val_a(I_a,"E") = \text{more}\\ \quad \wedge val_a(I_a,"F") = \text{more}\\ \textbf{then}\\ \quad \text{true}\\ \textbf{else}\\ \quad val_a(I_a,"E") = val_a(I_a,"F")\\ \textbf{fi}\end{array} \tag{A.196}$$

We can prove the soundness of this definition:

Lemma 29 *Soundness of strong abstraction of equality*

$$(\forall I_c, E, F)\Big(val_c(I_c,"E = F") \rightarrow val_a(abs_n(I_c), Abstract("E = F", weak, n))\Big) \tag{A.197}$$

Proof:

We will demonstrate that the left hand side is false for any given I_c, E ad F, assuming that the right hand side is false.

The right hand side of (A.197) can be reduced by expanding the Abstract function and the val_a function as follows:

$$\begin{array}{l}\textbf{\textit{if}}\\ \quad val_a(abs_n(I_c), Abstract("E",, n)) = more\\ \quad \wedge val_a(abs_n(I_c), Abstract("F",, n)) = more\\ \textbf{\textit{then}}\\ \quad true\\ \textbf{\textit{else}}\\ \quad val_a(abs_n(I_c), Abstract("E",, n))\\ \qquad = val_a(abs_n(I_c), Abstract("F",, n))\\ \textbf{\textit{fi}}\end{array} \tag{A.198}$$

If we assume that (A.198) evaluates to false, it means that the following assertions hold:

$$val_a(abs_n(I_c), Abstract("E",_, n)) \neq more \tag{A.199}$$

$$val_a(abs_n(I_c), Abstract("F",_, n)) \neq more \tag{A.200}$$

$$val_a(abs_n(I_c), Abstract("E",_, n)) \neq val_a(abs_n(I_c), Abstract("F",_, n)) \tag{A.201}$$

By applying Theorem 5; (A.199), (A.200) and (A.201) can be simplified as follows:

$$abs_n(val_c(I_c,"E")) \neq more \tag{A.202}$$
$$abs_n(val_c(I_c,"F")) \neq more \tag{A.203}$$
$$abs_n(val_c(I_c,"E")) \neq abs_n(val_c(I_c,"F")) \tag{A.204}$$

(A.203) and (A.203) imply that $val_c(I_c,"E") < n$ *and* $val_c(I_c,"E") < n$*. Therefore, the abstract value and the concrete value of these expressions are equal. (A.204) can be simplified as follows:*

$$val_c(I_c,"E") \neq val_c(I_c,"F")$$

That is the negation of the left hand side of (A.197) where the val_c *function has been expanded.*

Q.E.D.

A.3.4.2 Abstracting "Lower than"

The "<" operator is abstracted with weakening into the $<_{a,n,weak}$ operator. Therefore, the *Abstract* function is defined on $<$ as follows:

$$\begin{aligned} &Abstract("E < F", weak, n) =_{def} \\ &\quad Abstract("E", weak, n) <_{a,n,weak} Abstract("F", weak, n) \end{aligned} \tag{A.205}$$

The semantics of $<_{a,n,weak}$ is defined by the val_a function as follows:

$$\begin{aligned} &val_a(I_a,"E <_{a,n,weak} F") =_{def} \\ &\quad \textbf{if } val_a(I_a,"F") = \text{more } \textbf{then} \\ &\quad\quad \text{true} \\ &\quad \textbf{elseif } val_a(I_a,"E") = \text{more } \textbf{then} \\ &\quad\quad \text{false} \\ &\quad \textbf{else} \\ &\quad\quad val_a(I_a,"E") < val_a(I_a,"F") \\ &\quad \textbf{fi} \end{aligned} \tag{A.206}$$

We can prove the soundness of this definition:

Lemma 30 *Soundness of strong abstraction of "<"*

$$(\forall I_c, E, F)\Big(val_c(I_c, "E < F") \rightarrow val_a(abs_n(I_c), Abstract("E < F", weak, n))\Big) \quad \text{(A.207)}$$

Proof:

We will demonstrate that the left hand side is false for any given I_c, E ad F, assuming that the right hand side is false.

The right hand side of (A.207) can be reduced by expanding the Abstract function and the val_a function as follows:

$$\begin{array}{l} \textbf{\textit{if}}\ val_a(abs_n(I_c), Abstract("F", _, n)) = more\ \textbf{\textit{then}} \\ \quad true \\ \textbf{\textit{elseif}}\ val_a(abs_n(I_c), Abstract("E", _, n)) = more\ \textbf{\textit{then}} \\ \quad false \\ \textbf{\textit{else}} \\ \quad val_a(abs_n(I_c), Abstract("E", _, n)) \\ \qquad < val_a(abs_n(I_c), Abstract("F", _, n)) \\ \textbf{\textit{fi}} \end{array} \quad \text{(A.208)}$$

If we assume that (A.208) evaluates to false, it means that the following assertion holds:

$$val_a(abs_n(I_c), Abstract("F", _, n)) \neq more \quad \text{(A.209)}$$

By applying Theorem 5; (A.209) can be simplified as follows:

$$abs_n(val_c(I_c, "F")) \neq more$$

This formula implies that:

$$val_c(I_c, "F") \leq n \quad \text{(A.210)}$$

We now end up by showing that the right hand side of (A.207) is false on two cases taken from the second condition of (A.208).

- ***Case 1:*** $val_a(abs_n(I_c), Abstract("E", _, n)) = more$
  The case condition can be simplified by applying Theorem 5 as follows:

$$abs_n(val_c(I_c, "E")) = more$$

This formula expresses that

$$val_c(I_c, "E") > n \tag{A.211}$$

(A.210) and (A.211) imply the negation of the right hand side of (A.207) where the val_c function has been expanded.

- ***Case 2:*** $val_a(abs_n(I_c), Abstract("E", _, n)) \neq more$
  In this case, due to the fact that (A.208) is false by hypothesis, we also have the following assertion:

$$val_a(abs_n(I_c), Abstract("E", _, n)) \tag{A.212}$$
$$\geq val_a(abs_n(I_c), Abstract("F", _, n)) \tag{A.213}$$

The case condition and (A.213) can be simplified by applying Theorem 5 into the following assertions:

$$abs_n(val_c(I_c, "E")) \neq more \tag{A.214}$$
$$abs_n(val_c(I_c, "E")) \geq abs_n(val_c(I_c, "F")) \tag{A.215}$$

(A.210) and (A.214) imply that $val_c(I_c, "E") \leq n$ and $val_c(I_c, "F") \leq n$. Due to this, the abstraction of these values are therefore equal to their concrete value, wo that (A.215) can be reduced into:

$$val_c(I_c, "E") \geq val_c(I_c, "F")$$

That is the negation of the right hand side of (A.207) where the val_c function has been expanded.

Q.E.D.

A.3.4.3 Abstracting "Lower or Equal to"

The "$\leq$" operator is abstracted with weakening into the $\leq_{a,n,weak}$ operator. Therefore, the *Abstract* function is defined on $\leq$ as follows:

$$Abstract("E \leq F", weak, n) =_{def}$$
$$Abstract("E", weak, n) \leq_{a,n,weak} Abstract("F", weak, n) \tag{A.216}$$

The semantics of $\leq_{a,n,weak}$ is defined by the val_a function as follows:

$$
\begin{array}{l}
val_a(I_a, "E \leq_{a,n,weak} F") =_{def} \\
\quad \textbf{if } val_a(I_a, "F") = \text{more } \textbf{then} \\
\quad\quad \text{true} \\
\quad \textbf{elseif } val_a(I_a, "E") = \text{more } \textbf{then} \\
\quad\quad \text{false} \\
\quad \textbf{else} \\
\quad\quad val_a(I_a, "E") \leq val_a(I_a, "F") \\
\quad \textbf{fi}
\end{array}
\tag{A.217}
$$

We can prove the soundness of this definition:

Lemma 31 *Soundness of strong abstraction of "$\leq$"*

$$
\begin{array}{l}
(\forall I_c, E, F)\Big(val_c(I_c, "E \leq F") \\
\qquad \rightarrow val_a(abs_n(I_c), Abstract("E \leq F", weak, n))\Big)
\end{array}
\tag{A.218}
$$

Proof:

We will demonstrate that the left hand side is false for any given I_c, E ad F, assuming that the right hand side is false.

The right hand side of (A.207) can be reduced by expanding the Abstract function and the val_a function as follows:

$$
\begin{array}{l}
\textbf{\textit{if}}\ val_a(abs_n(I_c), Abstract("F", _, n)) = more\ \textbf{\textit{then}} \\
\quad true \\
\textbf{\textit{elseif}}\ val_a(abs_n(I_c), Abstract("E", _, n)) = more\ \textbf{\textit{then}} \\
\quad false \\
\textbf{\textit{else}} \\
\quad val_a(abs_n(I_c), Abstract("E", _, n)) \\
\qquad \leq val_a(abs_n(I_c), Abstract("F", _, n)) \\
\textbf{\textit{fi}}
\end{array}
\tag{A.219}
$$

If we assume that (A.219) evaluates to false, it means that the following assertion holds:

$$
val_a(abs_n(I_c), Abstract("F", _, n)) \neq more \tag{A.220}
$$

By applying Theorem 5; (A.220) can be simplified as follows:

$$abs_n(val_c(I_c, "F")) \neq more$$

This formula implies that:

$$val_c(I_c, "F") \leq n \tag{A.221}$$

We now end up by showing that the right hand side of (A.218) is false on two cases taken from the second condition of (A.219).

- ***Case 1:*** $val_a(abs_n(I_c), Abstract("E", _, n)) = more$
  The case condition can be simplified by applying Theorem 5 as follows:

  $$abs_n(val_c(I_c, "E")) = more$$

  This formula expresses that

  $$val_c(I_c, "E") > n \tag{A.222}$$

  (A.221) and (A.222) imply the negation of the right hand side of (A.218) where the val_c function has been expanded.

- ***Case 2:*** $val_a(abs_n(I_c), Abstract("E", _, n)) \neq more$
  In this case, due to the fact that (A.219) is false by hypothesis, we also have the following assertion:

  $$\begin{aligned} &val_a(abs_n(I_c), Abstract("E", _, n)) \\ &\quad > val_a(abs_n(I_c), Abstract("F", _, n)) \end{aligned} \tag{A.223}$$

  The case condition and (A.223) can be simplified by applying Theorem 5 into the following assertions:

  $$abs_n(val_c(I_c, "E")) \neq more \tag{A.224}$$

  $$abs_n(val_c(I_c, "E")) > abs_n(val_c(I_c, "F")) \tag{A.225}$$

  (A.221) and (A.224) imply that $val_c(I_c, "E") \leq n$ and $val_c(I_c, "F") \leq n$. Due to this, the abstraction of these values are therefore equal to their concrete value, wo that (A.225) can be reduced into:

  $$val_c(I_c, "E") > val_c(I_c, "F")$$

  That is the negation of the right hand side of (A.218) where the val_c function has been expanded.

Q.E.D.

A.3.4.4 Soundnes of the Weak Abstraction of Assertion

Theorem 8 *Soundness of weak abstraction of assertions on integers*

$$(\forall I_c, E, F)\Big(val_c(I_c, E) \quad \rightarrow \quad val_a(abs_n(I_c), Abstract(E, weak, n))\Big) \tag{A.226}$$

Proof:

According to the grammar of assertions, we have to consider three cases:

- ***E is an equality***
  The demonstration goal is then:

$$(\forall I_c, E, F)\Big(val_c(I_c, "E = F") \rightarrow val_a(abs_n(I_c), Abstract("E = F", weak, n))\Big)$$

  This is proven in Lemma 29.

- ***E is a "lower than"***
  The demonstration goal is then:

$$(\forall I_c, E, F)\Big(val_c(I_c, "E < F") \rightarrow val_a(abs_n(I_c), Abstract("E < F", weak, n))\Big)$$

  This is proven in Lemma 30.

- ***E is a "lower or equal to"***
  The demonstration goal is then:

$$(\forall I_c, E, F)\Big(val_c(I_c, "E \leq F") \rightarrow val_a(abs_n(I_c), Abstract("E \leq F", weak, n))\Big)$$

  This is proven in Lemma 31.

Q.E.D.

A.4 Input Files for the e-Signature System

This section presents the input files representing the e-signature system in the input language of CONCHITA . These input files were generated in the course of the confidentiality analysis presented in Section 4.6.

The following input file represents the very first model:

```
%This is a file for analysing the signature system
%here, we explore an alternative model
%for representing information flow

agent EID{
password:int
}

agent Signer{
}

association Represent{
eid(0:1):EID
signer(1:1):Signer
}

agent SignBox{
}

association EnteredPassword{
into(0:1):SignBox
val:int
}

association Regular{
s(0:1):SignBox
}

association Inserted{
what(0:1):EID
into(0:1):SignBox
}

association AgreesOnDisplayedFile{
```

```
who(0:):Signer
where(0:):SignBox
}

association at{
who(0:1):Signer
where(0:1):SignBox
}

dominvar
(forall s:SignBox)
([] Regular<s:s>) or ([]!Regular<s:s>)
dominvar
(forall i:Signer)
(exists e:EID) [] Represent<eid:e signer:i>

Signer:[bobby]
int:2
EID:1
SignBox:1
AgreesOnDisplayedFile:1
Inserted:1
Represent:1
Regular:1
at:1

goal confidentiality[EID password]
(forall e:EID a:Agent)
not Represent<eid:e signer:a>
& not a=e
& not Regular<s:a>
=> full-confidential-now(a,e.password)

goal achieve[password entered into sign box
          when agreement on displayed document]
(forall s:SignBox e:EID i:Signer)
AgreesOnDisplayedFile<who:i where:s>
& Inserted<what:e into:s>
& Represent<eid:e signer:i>
& at<who:i where:s>
=> (o (EnteredPassword<into:s val:e.password>
& Inserted<what:e into:s>
& at<who:i where:s>))
```

The first model proved to include a confidentiality violation. It is prevented by adding the new goal *maintain[Entered Password Confidential]*. This gives rise to the second model represented by the following input file:

```
%This is a file for analysing the signature system
%here, we explore an alternative model
%for representing information flow

agent EID{
password:int
}

agent Signer{
}

association Represent{
eid(0:1):EID
signer(1:1):Signer
}

agent SignBox{
}

association EnteredPassword{
into(0:1):SignBox
val:int
}

association Regular{
s(0:1):SignBox
}

association Inserted{
what(0:1):EID
into(0:1):SignBox
}

association AgreesOnDisplayedFile{
who(0:):Signer
where(0:):SignBox
}

association at{
```

```
who(0:1):Signer
where(0:1):SignBox
}

dominvar
(forall s:SignBox)
([] Regular<s:s>) or ([]!Regular<s:s>)
dominvar
(forall i:Signer)
(exists e:EID) [] Represent<eid:e signer:i>

Signer:[bobby]
int:2
EID:1
SignBox:1
AgreesOnDisplayedFile:1
Inserted:1
Represent:1
Regular:1
at:1

goal confidentiality[EID password]
(forall e:EID a:Agent)
not Represent<eid:e signer:a>
& not a=e
& not Regular<s:a>
=> full-confidential-now(a,e.password)

goal achieve[password entered into regular sign box
             when agreement on displayed document]
(forall s:SignBox e:EID i:Signer)
AgreesOnDisplayedFile<who:i where:s>
& Inserted<what:e into:s>
& Represent<eid:e signer:i>
& at<who:i where:s>
=> (o (EnteredPassword<into:s val:e.password>
& Inserted<what:e into:s>
& at<who:i where:s>))

goal confidentiality[SngBox Password]
(forall s:SignBox a:Agent)
not a=s
& not at<who:a where:s>
=> full-confidential-now(a,EnteredPassword<into:s>.val)
```

Unfortunately, another confidentiality violation was discovered on the second model. This new violation is resolved by allowing SignBoxes to know the password entered into them. This gives rise to the third model represented by the following input file:

```
%This is a file for analysing the signature system
%here, we explore an alternative model
%for representing information flow

agent EID{
password:int
}

agent Signer{
}

association Represent{
eid(0:1):EID
signer(1:1):Signer
}

agent SignBox{
}

association EnteredPassword{
into(0:1):SignBox
val:int
}

association Regular{
s(0:1):SignBox
}

association Inserted{
what(0:1):EID
into(0:1):SignBox
}

association AgreesOnDisplayedFile{
who(0:):Signer
```

```
where(0:):SignBox
}

association at{
who(0:1):Signer
where(0:1):SignBox
}

dominvar
(forall s:SignBox)
([] Regular<s:s>) or ([]!Regular<s:s>)
dominvar
(forall i:Signer)
(exists e:EID) [] Represent<eid:e signer:i>

Signer:[bobby]
int:2
EID:1
SignBox:1
AgreesOnDisplayedFile:1
Inserted:1
Represent:1
Regular:1
at:1

goal confidentiality[EID password]
(forall e:EID a:Agent)
not Represent<eid:e signer:a>
& not a=e
& not Regular<s:a>
=> full-confidential-now(a,e.password)

goal achieve[password entered into sign box
             when agreement on displayed document]
(forall s:SignBox e:EID i:Signer)
AgreesOnDisplayedFile<who:i where:s>
& Inserted<what:e into:s>
& Represent<eid:e signer:i>
& at<who:i where:s>
=> (o (EnteredPassword<into:s val:e.password>
& Inserted<what:e into:s>
& at<who:i where:s>))

goal confidentiality[SngBox Password]
```

```
(forall s:SignBox a:Agent)
not a=s
& not at<who:a where:s>
& not Inserted<what:a into:s>
=> full-confidential-forever(a,EnteredPassword<into:s>.val)
```

Again, the new model revealed another confidentiality violation. This time, the signer enters her password into a non-genuine SignBox, which is not allowed to know the password. In order to prevent this confidentiality violation, we add a security goal stating that the password should not be entered into a non-genuine SignBox. This new goal conflicts with a functional goal. The functional goal is therefore weakened in order to remove de conflict. This gives rise to the fourth model represented by the following input file:

```
%This is a file for analysing the signature system
%here, we explore an alternative model
%for representing information flow

agent EID{
password:int
}

agent Signer{
}

association Represent{
eid(0:1):EID
signer:Signer
}

agent SignBox{
}

association EnteredPassword{
into(0:1):SignBox
val:int
}

association Regular{
s(0:1):SignBox
```

```
}

association Inserted{
what(0:1):EID
into(0:1):SignBox
}

association ForwardedPassword{
password:int
card(0:1):EID
}

association AgreesOnDisplayedFile{
who(0:):Signer
where(0:):SignBox
}

association at{
who(0:1):Signer
where(0:1):SignBox
}

dominvar
(forall s:SignBox)
([] Regular<s:s>) or ([]!Regular<s:s>)
dominvar
(forall i:Signer)
(exists e:EID) [] Represent<eid:e signer:i>

dominvar
(forall s:SignBox)
EnteredPassword<into:s> => at<where:s>

Signer:[bobby]
int:2
EID:1
SignBox:1
AgreesOnDisplayedFile:1
Inserted:1
Represent:1
Regular:1
at:1

goal confidentiality[EID password]
```

```
(forall e:EID a:Agent)
not Represent<eid:e signer:a>
& not a=e
& not Regular<s:a>
=> full-confidential-forever(a,e.password)

goal achieve[password entered into regular sign box
             when agreement on displayed document]
(forall s:SignBox e:EID i:Signer)
AgreesOnDisplayedFile<who:i where:s>
& Inserted<what:e into:s>
& Represent<eid:e signer:i>
& Regular<s:s>
& at<who:i where:s>
=> (o (EnteredPassword<into:s val:e.password>
& Inserted<what:e into:s>
& at<who:i where:s>))

goal avoid[password entered into non regular sign box]
(forall s:SignBox)
not Regular<s:s>
=> not EnteredPassword<into:s>

goal confidentiality[SngBox Password]
(forall s:SignBox a:Agent)
not a=s
& not at<who:a where:s>
& not Inserted<what:a into:s>
=> full-confidential-forever(a,EnteredPassword<into:s>.val)
```

Following the confidentiality analysis of the e-signature system, there is a discussion in Section 4.6.3 that discusses over a variant of the previous model. This variant does not have a security goal prescribing that the password should not be entered into a non-genuine SignBox. The following input file represents this variant:

```
%This is a file for analysing the signature system
%here, we explore an alternative model
%for representing information flow

agent EID{
password:int
```

```
}

agent Signer{
}

association Represent{
eid(0:1):EID
signer:Signer
}

agent SignBox{
}

association EnteredPassword{
into(0:1):SignBox
val:int
}

association Regular{
s(0:1):SignBox
}

association Inserted{
what(0:1):EID
into(0:1):SignBox
}

association ForwardedPassword{
password:int
card(0:1):EID
}

association AgreesOnDisplayedFile{
who(0:):Signer
where(0:):SignBox
}

association at{
who(0:1):Signer
where(0:1):SignBox
}

dominvar
(forall s:SignBox)
```

```
([] Regular<s:s>) or ([]!Regular<s:s>)
dominvar
(forall i:Signer)
(exists e:EID) [] Represent<eid:e signer:i>

dominvar
(forall s:SignBox)
EnteredPassword<into:s> => at<where:s>

Signer:[bobby]
int:2
EID:1
SignBox:1
AgreesOnDisplayedFile:1
Inserted:1
Represent:1
Regular:1
at:1

goal confidentiality[EID password]
(forall e:EID a:Agent)
not Represent<eid:e signer:a>
& not a=e
& not Regular<s:a>
=> full-confidential-forever(a,e.password)

goal achieve[password entered into regular sign box
             when agreement on displayed document]
(forall s:SignBox e:EID i:Signer)
AgreesOnDisplayedFile<who:i where:s>
& Inserted<what:e into:s>
& Represent<eid:e signer:i>
& Regular<s:s>
& at<who:i where:s>
=> (o (EnteredPassword<into:s val:e.password>
& Inserted<what:e into:s>
& at<who:i where:s>))

goal confidentiality[SngBox Password]
(forall s:SignBox a:Agent)
not a=s
& not at<who:a where:s>
& not Inserted<what:a into:s>
=> full-confidential-forever(a,EnteredPassword<into:s>.val)
```

Appendix B

Annex of Chapter 5

B.1 Soundness of the Local Axioms with Respect to the Completeness Criterion

This section presents the proof that enforcing all the meta-constraints of the trustworthiness method ensures that, once the requirements model is finished, all security-critical goals are securely enforced. This section first recalls the definitions of the concepts involved in this proof, the axioms and the definition of *securely enforced*. Throughout this section, some parameters of predicate will be denoted by the "_" symbol, this means that they are implicitely existentially quantified with the existential quantification being nested just around the concerned predicate. The base concepts manipulated are defined as follows:

Wish(ag, G) **iff** agent ag is the owner of an asset that goal G requires to protect.

Trustworthy(ag, G, w) **iff** from agent w's viewpoint, agent ag will enforce those requirements or assumptions assigned to it which contribute to the security goal G.

Distrust(O, R, v) **iff** leaf goal R contributes to a security goal wished by v and obstacle condition O is equivalent to: the agent responsible for requirement R is not trustworthy from the viewpoint of v. If condition O is faflse, then the agent responsible for the requirement R is trustworthy from the viewpoint of v.

From these base concepts and concepts from the KAOS meta-model, we define the folowing derived concepts:

CounterMeasure$(CG, LG, O, Gref)$ **iff** goal CG is a countermeasure to the untrustworthiness O of the assignment of leaf goal LG, that enforces the higher-level reference goal $Gref$. This is formally defined as follows: $CG \models O \rightarrow LG \vee Gref$. We sometimes add a fifth argument to this predicate to denote the viewpoint from which O is an untrustworthiness of LG.

DirectlyContributesTo$(G1, G2, v)$ **iff** (a) $G1$ refines $G2$ or (b) $G1$ is a countermeasure, from v's viewpoint, to an untrustworthy assignment of leaf goal $G2$ or of another leaf goal through enforcement of reference goal $G2$. This is formally defined as follows: Refines$(G_1, G_2) \vee$ CounterMeasure$(G_1, _, _, G_2, v) \vee$ CounterMeasure$(G_1, G_2, _, _, v)$

ContributesTo$(G1, G2, w)$ **iff** goal $G1$ directly contributes to goal $G2$ or to some goal G' that contributes to goal $G2$.

DominatesContribution$(G1, G2, w)$ **iff** from the viewpoint of agent w, $G2$ contributes to $G1$ and if $G2$ contributes to some goal then this goal either contributes to $G1$ or $G1$ contributes to it.

Our method guarantees that all critical security goals are securely enforced from all concerned viewpoints. This is formally defined as follows:

$$\begin{array}{l}(\forall G : Goal, w : Agent) \\ \quad \text{Wish}(ag, G) \\ \quad \rightarrow \Big(\wedge_{\{R | Trustworthy(Resp(R), G, w)\}} R \models G \Big)\end{array} \tag{B.1}$$

This property is enforced through the enforcement of four axioms on the requirements model. These axioms can be enforced via a set of local reasoning on fragments of the requirements model. This facilitates the elaboration of requirements model enforcing the completeness property (B.1). The four axioms are recalled here.

Our first axiom on countermeasures defines the conditions under which an untrustworthiness of responsibility assignment obstacle should be introduced in the model:

$$\begin{array}{l}(\forall ag, w : Agent, LG : LeafGoal, G : Goal) \\ \quad \text{Wish}(w, G) \wedge \text{ContributesTo}(LG, G, w) \\ \quad \wedge \text{ResponsibleFor}(ag, LG) \wedge \neg Trustworthy(ag, G, w) \\ \quad \rightarrow (\exists O : Obstacle) \text{Distrust}(O, LG, w)\end{array}$$

Our second axiom defines the conditions under which a countermeasure to an untrustworthy assignment should be introduced so as to ensure a higher-

level reference goal, basically, whenever there is an untrustworthiness:

$$(\forall LG : LeafGoal, w : Agent, O : Obstacle)$$
$$\text{Distrust}(O, LG, w)$$
$$\rightarrow (\exists GC, Gref : Goal)\text{CounterMeasure}(GC, LG, O, Gref)$$

The reference goal must be on all paths from the obstructed leaf goal to the critical security goal it contributes to. Our third axiom ensures that the reference goal dominates the contribution of the obstructed goal:

$$(\forall LG : LeafGoal, CG, Gref : Goal, w : Agent, O : Obstacle)$$
$$\text{Distrust}(O, LG, w) \wedge \text{CounterMeasure}(GC, O, LG, Gref)$$
$$\rightarrow \text{DominatesContribution}(Gref, LG, w)$$

Our fourth axiom ensures that the countermeasure is not taken too high in the goal graph, so that the critical security goal is enforced, not the goals it contributes to:

$$(\forall LG : LeafGoal, CG, Gref, Gcritical : Goal, w : Agent, O : Obstacle)$$
$$\text{Wish}(w, Gcritical) \wedge \text{ContributesTo}(LG, Gcritical, w)$$
$$\text{Distrust}(LG, O, w) \wedge \text{CounterMeasure}(CG, O, LG, Gref)$$
$$\rightarrow \text{ContributesTo}(Gref, Gcritical, w)$$

Our demonstration is structured into two lemmas and one theorem. The first lemma demonstrates some annex property of the ContributesTo and DominatesContribution relations. The second one proves that all goals enforce a certain property corresponding to a relaxation of the criterion. The theorem concludes the proof by showing how the second lemma implies that the completeness criterion is enforced.

Lemma 32 *Let be the goals g_1, g_2 and g_3 and agent v such that:*

$$ContributesTo(g_1, g_2, v) \tag{B.2}$$
$$DominatesContribution(g_3, g_1, v) \tag{B.3}$$
$$ContributesTo(g_2, g_3, v) \tag{B.4}$$

The following assertion is true:

$$DominatesContribution(g_3, g_2, v) \tag{B.5}$$

Proof:

First, we expand the proof goal (B.5) according to the definition of DominatesContribution into the following two proof goals:

$$(\forall g_5)\, ContributesTo(g_2, g_5, v) \\ \quad \rightarrow ContributesTo(g_5, g_3, v) \vee ContributesTo(g_3, g_5, v) \quad \text{(B.6)}$$

$$ContributesTo(g_2, g_3, v) \quad \text{(B.7)}$$

Equation (B.7) is trivially true from (B.4). By extentionality, we assume a given g_5. We will show that (B.6) holds for this g_5. We can assume that the premise of (B.6) holds for this g_5, we are going to prove the consequent of (B.6).

With the premise of (B.6), with (B.2) and by transitivity of the relation ContributesTo, we have:

$$ContributesTo(g_1, g_5, v) \quad \text{(B.8)}$$

We expand (B.3) into the following:

$$(\forall g_4)\, ContributesTo(g_1, g_4, v) \\ \quad \rightarrow ContributesTo(g_4, g_3, v) \vee ContributesTo(g_3, g_4, v) \quad \text{(B.9)}$$

$$ContributesTo(g_1, g_3, v) \quad \text{(B.10)}$$

We can instantiate g_4 in (B.9) to g_5. The premice of this instantiation is true, thanks to (B.8), so that we have:

$$ContributesTo(g_5, g_3, v) \vee ContributesTo(g_3, g_5, v) \quad \text{(B.11)}$$

That trivially implies the consequent of (B.6).

Q.E.D.

Lemma 33 *Let v be an agent taken as a viewpoint and g_{crit} a critical security goal wished by v:*

$$Wish(v, g_{crit}) \quad \text{(B.12)}$$

If (a) all requirements assigned to trustworthy agents are enforced, and (b) the four axioms presented above are enforced and (c) the requirements model is finished, that is: every goal is either refined or assigned to an agent, then every goal contributing to g_{crit} is either enforced or there is a dominator

of this goal that is enforced and that contributes to g_{crit}. The condition (a) is formally defined as follows:

$$\begin{aligned}&(\forall r\colon Requirement\ ag\colon Agent)\\&\quad Trustworthy(ag, g_{crit}, v)\\&\quad \wedge ResponsibleFor(ag, r)\\&\quad \wedge ContributesTo(r, g_{crit}, v)\\&\quad \rightarrow r\end{aligned} \tag{B.13}$$

The assertion we have to prove is formally defined as follows:

$$\begin{aligned}&(\forall g_2\colon Goal)\\&\quad ContributesTo(g_2, g_{crit}, v)\\&\quad \rightarrow g_2 \vee (\exists g_3\colon Goal)\\&\qquad DominatesContribution(g_3, g_2, v)\\&\qquad \wedge ContributesTo(g_3, g_{crit}, v)\\&\qquad \wedge g_3\end{aligned}$$

Proof:

The proof proceeds by induction over the structure of the goal graph, namely, over the refinement relation and over the countermeasure relation between untrustworthy requirements and countermeasures. This covers all the goals of the goal graph. This induction is well-founded because the goal graph is a finite directed acyclic graph. The induction gives rise to three cases to proof:

- ***Case 1:*** *nothing contributes to g_2, therefore it is a requirement under the responsibility of an agent (base case)*
- ***Case 2:*** *g_2 is refined into two goals (inductive case). We do not consider the case where g_2 is refined into more than two goals because they are only generalization of the two-goals case.*
- ***Case 3:*** *g_2 untrustworthily assigned to an agent and there is a countermeasure goal contributing to it. (inductive case)*

We now detail the four cases of the proof in detail:

- ***Case 1:*** $\neg Refines(_, g_2) \wedge \neg Distrust(_, g_2, v)$

In this case, we prove the following:

$$ContributesTo(g_2, g_{crit}, v) \rightarrow g_2$$

Under the finished model assumption*, there is an agent ag that is responsible for g_2: $ResponsibleFor(ag, g_2)$. If we assume that $ContributesTo(g_2, g_{crit}, v)$, the first axiom ensures that the assertion $Trustworthy(ag, g_{crit}, v)$ holds because there is no distrust condition. Therefore, the goal proof presented here is trivially true by (B.13)*

- ***Case 2:*** *In this case, g_2 is refined.*

  We suppose that there are only two goals in the refinement of g_2, namely: g_l and g_r (l and r stand for left *and* right*). We have the following inductive hypothesis:*

$$\begin{aligned} & g_l \vee (\exists g_{l3}\colon Goal) \\ & \quad DominatesContribution(g_{l3}, g_l, v) \\ & \quad \wedge ContributesTo(g_{l3}, g_{crit}, v) \\ & \quad \wedge g_{l3} \end{aligned}$$

$$\begin{aligned} & g_r \vee (\exists g_{r3}\colon Goal) \\ & \quad DominatesContribution(g_{r3}, g_r, v) \\ & \quad \wedge ContributesTo(g_{r3}, g_{crit}, v) \\ & \quad \wedge g_{r3} \end{aligned}$$

  We envision four cases on these hypotheses, based on the disjuncts appearing in both of them. They are as follows:

  - *$g_l \wedge g_r$, this is the case 2.1 below.*
  - *The second disjunct of the first hypothesis and the first disjunct of the second hold. This is the case 2.2 below.*
  - *The second disjunct of the second hypothesis holds the first disjunct of the first hold. This case is symmetric to the previous one and is therefore skipped.*
  - *The second disjunct of both hypothesis hold. This is the case 2.3 below.*

  The proof of the three relevant cases are detailed here.

  - ***Case 2.1:*** $g_l \wedge g_r$

    In this case, the semantics of refinement ensures that g_2 holds.

– ***Case 2.2:*** *in this case, there is a goal named* g_{l3} *such that:*

$$\begin{aligned} &DominatesContribution(g_{l3}, g_l, v) \\ &\wedge ContributesTo(g_{l3}, g_{crit}, v) \\ &\wedge g_{l3} \end{aligned} \tag{B.14}$$

and we also have that g_r *holds.*
We envision four cases about the relative position of g_{l3}, g_2 *and* g_l *in the goal graph:*

* ***Case 2.2.1:*** $g_{l3} = g_l$ *In this case,* g_l *holds. On the other hand, we have that* g_r *holds. Thanks to the semantics of refinements, this finishes the proof that* g_2 *holds.*
* ***Case 2.2.2:*** $g_{l3} = g_2$
In this case, g_2 *holds because of the last conjunct of the case 2.2 condition.*
* ***Case 2.2.3:*** $ContributesTo(g_2, g_{l3}) \wedge g_2 \neq g_{l3}$
In this case, we prove the second disjunct of the proof goal. We show that g_{l3} *is an appropriate value for the existentially quantified variable* g_3*. In order to do this, we have to prove the following:*

$$\begin{aligned} &DominatesContribution(g_{l3}, g_2, v) \\ &\wedge ContributesTo(g_{l3}, g_{crit}, v) \\ &\wedge g_{l3} \end{aligned}$$

The first conjunct is true thanks to Lemma 32 applied to the first conjunct of (B.14), to $ContributesTo(g_l, g_2, v)$*, because* g_l *refines* g_2*, and to* $ContributesTo(g_2, g_{l3})$ *that comes from the condition of case 2.2.3. The two last conjuncts are true because of (B.14). This finishes the proof of this case.*
* ***Case 2.2.4:*** $\neg ContributesTo(g_2, g_{l3}) \wedge g_{l3} \neq g_2 \wedge g_{l3} \neq g_l$
This case is forbidden because $ContributesTo(g_l, g_{l3}, v)$ *is true from the first conjunct of (B.14),* $ContributesTo(g_l, g_2, v)$ *from the fact that* g_l *refines* g_2 *and because ContributesTo is reflexive, antisymmetric and transitive. This finishes the proof.*

– ***Case 2.3:*** *In this case, there are some goals* g_{l3} *and* g_{r3} *such that:*

$$\begin{aligned} &DominatesContribution(g_{l3}, g_l, v) \\ &\wedge ContributesTo(g_{l3}, g_{crit}, v) \\ &\wedge g_{l3} \end{aligned} \tag{B.15}$$

$$DominatesContribution(g_{r3}, g_r, v) \\ \wedge ContributesTo(g_{r3}, g_{crit}, v) \\ \wedge g_{r3} \quad \text{(B.16)}$$

We envision several cases based on the relative position of g_{l3} and g_{r3} in the goal graph. The cases are described below:

- ***Case 2.3.1:*** $g_{l3} = g_l \wedge g_{r3} = g_r$
  In this case, we have $g_l \wedge g_r$. This finishes the proof that g_2 holds thanks to the semantics of refinements.
- ***Case 2.3.2:*** $ContributesTo(g_2, g_{r3}, v)$
  In this case, we finish the proof by showing that the second disjunct of the proof goal holds with $g_3 := g_{r3}$. We have to show that:

  $$DominatesContribution(g_2, g_{r3}, v) \\ \wedge ContributesTo(g_{r3}, g_{crit}, v) \\ \wedge g_{r3}$$

  The first conjunct is true thanks to Lemma 32 applied to the first conjunct of (B.16), to $ContributesTo(g_r, g_2, v)$, because g_r refines g_2, and to $ContributesTo(g_2, g_{r3}, v)$ that comes from the condition of case 2.3.2. The two last conjuncts are true because of (B.16). This finishes the proof of this case.
- ***Case 2.3.3:*** $ContributesTo(g_2, g_{l3}, v)$
  This case is similar to the previous one and is therefore skipped.

- ***Case 3:*** *in this case, g_2 is assigned to an agent, but the responsibility assignment is untrustworthy.*

  Because of the first axiom, there is thus an obstacle O that is the condition of untrustworthiness of the responsibility assignemnt:

  $$Distrust(O, g_2, v) \quad \text{(B.17)}$$

  Because of the second axiom, this untrustworthiness is resolved by a countermeasure goal g_c with reference goal g_{c3}:

  $$CounterMeasure(g_c, g_2, o, g_{c3}) \quad \text{(B.18)}$$

  Because of the third axiom, this reference goal dominates the contribution of g_2:

  $$DominatesContribution(g_{c3}, g_2, v) \quad \text{(B.19)}$$

Because of the fourth axiom and to (B.12) , the reference goal contributes to g_{crit}:

$$ContributesTo(c_{c3}, g_{crit}, v) \tag{B.20}$$

The inductive hypothesis applies on g_c, and the proof goal applies on g_2. They are recalled below. The inductive hypothesis is (B.21). The proof goal is actually an implication. The premice (B.21) is taken as hypothesis, the consequent (B.22) must be proven.

$$\begin{aligned} &ContributesTo(g_c, g_{crit}, v) \\ &\rightarrow g_c \vee (\exists g_4\colon Goal) \\ &\qquad DominatesContribution(g_4, g_c, v) \\ &\qquad \wedge ContributesTo(g_4, g_{crit}, v) \\ &\qquad \wedge g_4 \\ &ContributesTo(g_2, g_{crit}, v) \end{aligned} \tag{B.21}$$

$$\begin{aligned} &g_2 \vee (\exists g_3\colon Goal) \\ &\quad DominatesContribution(g_3, g_2, v) \\ &\quad \wedge ContributesTo(g_3, g_{crit}, v) \\ &\quad \wedge g_3 \end{aligned} \tag{B.22}$$

(B.23)

The premice of (B.21) is true because of (B.21) and to the fact that $ContributesTo(g_c, g_2, v)$. The latter is implied by the case condition because g_c is a countermeasure and as such contributes to g_2 by definition of DirectlyContributesTo. We therefore reduce the hypothesis (B.21) into (B.24).

$$\begin{aligned} &g_c \vee (\exists g_4\colon Goal) \\ &\quad DominatesContribution(g_4, g_c, v) \\ &\quad \wedge ContributesTo(g_4, g_{crit}, v) \\ &\quad \wedge g_4 \end{aligned} \tag{B.24}$$

We envision two cases based on the disjunction of (B.24):

- ***Case 3.1:*** *The first disjunct of (B.24) holds: g_c*
  Expanding CounterMeasure from (B.18) according to its definition yields the following expanded case condition:

$$g_c \wedge O \rightarrow g_2 \vee g_{c3}$$

This reduces to the following, because of the case condition:

$$O \rightarrow g_2 \vee g_{c3}$$

We envision three cases based on the last conjunct of (B.25):

* ***Case 3.1.1:*** $\neg O$
  In this case, as we have $ResponsibleFor(ag, g_2, v)$ *and under the hypothesis (B.13), we have that* g_2 *is true. That proves the first disjunct of (B.22). Hypothesis (B.13) applies because the distrust condition O is false, so that the responsible agent is trustworthy. This finishes the proof.*
* ***Case 3.1.2:*** g_2
  In this case, the proof is trivially finished because it is the first disjunct of (B.22).
* ***Case 3.1.3:*** g_{c3}
  In this case, we prove the second disjunct of (B.22) with $g_3 := g_{c3}$*. We have to prove the following:*

$$\begin{aligned} &DominatesContribution(g_{c3}, g_2, v) \\ &\wedge ContributesTo(g_{c3}, g_{crit}, v) \\ &\wedge g_{c3} \end{aligned} \tag{B.25}$$

  The first conjoint holds because of (B.19). The second holds because of (B.20). The third holds because of the condition of case 3.1.3. This finishes the proof of this case.

– ***Case 3.2:*** *The second disjunct of (B.24) holds, there is a* g_4 *suh that:*

$$\begin{aligned} &DominatesContribution(g_4, g_c, v) \\ &\wedge ContributesTo(g_4, g_{crit}, v) \\ &\wedge g_4 \end{aligned} \tag{B.26}$$

We envision three subcases about the relative position of g_4 *and* g_2*:*

* ***Case 3.2.1:*** $g_2 = g_4$
  In this case, the proof is trivially finished because the first disjunct of (B.22) is true.
* ***Case 3.2.2:*** $ContributesTo(g_2, g_4, v)$
  In this case, we prove the second disjunct of (B.22) with $g_3 := g_4$*. We have to prove:*

$$\begin{aligned} &DominatesContribution(g_2, g_4, v) \\ &\wedge ContributesTo(g_4, g_{crit}, v) \\ &\wedge g_4 \end{aligned} \tag{B.27}$$

The first conjunct is true thanks to Lemma 32 applied to the first conjunct of (B.26), to the case condition of case 3 saying that $ContributesTo(g_c, g_2, v)$ *and to the case condition of case 3.2.2. The two last conjuncts are true because of (B.26). This finishes the proof of this case.*

Q.E.D.

We now conclude our proof by the following theorem:

Theorem 9 *Soundness of the trustworthiness axioms*

Let be v *an agent and* g_{crit} *a critical security goal such that:*

$$Wish(v, g_{crit})$$

If (a) all requirements assigned to trustworthy agent are enforced, and (b) the four axioms presented above are enforced and (c) the requirements model is finished, that is: every goal is either refined or assigned to an agent, then g_{crit} *is enforced.*

Proof:

Thanks to Lemma 33, the following holds about all goals including g_{crit}*:*

$$\begin{aligned}
&(\forall g_2 \colon Goal) \\
&\quad ContributesTo(g_2, g_{crit}, v) \\
&\quad \rightarrow g_2 \vee (\exists g_3 \colon Goal) \\
&\qquad\qquad DominatesContribution(g_3, g_2, v) \\
&\qquad\qquad \wedge ContributesTo(g_3, g_{crit}, v) \\
&\qquad\qquad \wedge g_3
\end{aligned}$$

This also holds for g_{crit}*. As the ContributesTo relationship is reflexive, the antecedent of this assertion holds for* $g_2 := g_{crit}$*. We envision two cases based on the disjunct appearing in the consequent of the resulting assertion:*

- ***Case 1:*** *The first disjunct holds:* g_{crit} *This finishes the proof.*
- ***Case 2:*** *The second dinjunct holds:*

$$(\exists g_3 \colon Goal)$$

$$DominatesContribution(g_3, g_{crit}, v)$$
$$\wedge ContributesTo(g_3, g_{crit}, v)$$
$$\wedge g_3$$

As ContributesTo is antisymmetric, we have that $g_3 = g_{crit}$ because of the second and third conjunct. Because of the last conjunct, we have that g_3 holds. This finishes the proof.

Q.E.D.