# Méthode pour la spécification de responsabilité pour les logiciels : Modelisation, Tracabilité et Analyse de dysfonctionnements

Eduardo Sampaio Elesbao Mazza Sampaio Elesbao Mazza

UNIVERSITÉ DE GRENOBLE

# THÈSE

Pour obtenir le grade de

## DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel :

Présentée par

**Eduardo Sampaio Elesbao Mazza**

Thèse dirigée par **Marie-Laure Potet**
et codirigée par **Daniel Le Métayer**

préparée au sein de **VERIMAG**
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

# A Formal Framework for Specifying and Analyzing Liabilities Using Log as Digital Evidence

Thèse soutenue publiquement le ,
devant le jury composé de :

**Roland Groz**
Professeur, Laboratoire d'Informatique de Grenoble, Président
**Regine Laleau**
Professeur, Université Paris-Est Créteil, Rapporteur
**Gerardo Schneider**
Professeur, University of Gothenburg, Rapporteur
**Guillaume Dufay**
Consultant Sécurité, Trusted Labs, Examinateur
**Jean-Marc Jézéquel**
Professeur, Université de Rennes I, Examinateur
**Marie-Laure Potet**
Professeur, VERIMAG, Directeur de thèse
**Daniel Le Métayer**
Directeur de Recherche, INRIA, Co-Directeur de thèse

# Acknowledgments

I would like to express my immense gratitude to my two supervisors: Marie-Laure Potet and Daniel Le Métayer, whose knowledge, guidance and patience added to my working experience. I appreciate, specially, their assistance in the writing process which helped me to produce this document. I would like to thank also the other members of the project LISE to the assistance provided at other aspects of my research.

Besides my supervisors and collaborators, I would like to thanks my thesis committee: Regine Laleau, Gerardo Schneider, Guillaume Dufay, Jean-Marc Jézéquel and Roland Groz; for taking their time to read and evaluate my work, and also provide their critical opinion and interesting questions about the research.

I also thank the staff members of Verimag, specially to Sandrine Magnin and Christine Saunier, for their help with the administrative process. I thank some of my fellow PhD students: Jean Quilbeuf, Artur Pietrek and Balaji Raman. Each of them helped to make my time in the PhD program much more fun and interesting and they also provided support with the administrative process of my PhD while I was outside France.

I must say that without the immense love, support and encouragement of my wife Julia I would never have finished my thesis and for this I am eternally grateful. I would also like to thanks my family that, besides the geographical distance, also provided support during my PhD.

Finally, I recognize that this research would not be possible without the financial assistance of the ANR[1] through the project LISE (ANR-07-SESU-007).

[1] Agence Nationale de la Recherche

# Abstract

Despite the effort made to define methods for the design of high quality software, experience shows that failures of IT systems due to software errors remain very common and one must admit that even critical systems are not immune from that type of errors. One of the reasons for this situation is that software requirements are generally hard to elicit precisely and it is often impossible to predict all the contexts in which software products will actually be used. Considering the interests at stake, it is therefore of prime importance to be able to establish liabilities when damages are caused by software errors. Essential requirements to define these liabilities are (1) the availability of reliable evidence, (2) a clear definition of the expected behaviors of the components of the system and (3) the agreement between the parties with respect to liabilities. In this thesis, we address these problems and propose a formal framework to precisely specify and establish liabilities in a software contract. This framework can be used to assist the parties both in the drafting phase of the contract and in the definition of the architecture to collect evidence. Our first contribution is a method for the integration of a formal definition of digital evidence and liabilities in a legal contract. Digital evidence is based on distributed execution logs produced by "acceptable log architectures". The notion of acceptability relies on a formal threat model based on the set of potential claims. Another main contribution is the definition of an incremental procedure, which is implemented in the LAPRO tool, for the analysis of distributed logs.

# Abstract in French

Malgré les progrès importants effectués en matière de conception de logiciels et l'existence de méthodes de développement éprouvées, il faut reconnaître que les défaillances de systèmes causées par des logiciels restent fréquentes. Il arrive même que ces défaillances concernent des logiciels critiques et provoquent des dommages significatifs. Considérant l'importance des intérêts en jeu, et le fait que la garantie de logiciel "zéro défaut" est hors d'atteinte, il est donc important de pouvoir déterminer en cas de dommages causés par des logiciels les responsabilités des différentes parties. Pour établir ces responsabilités, un certain nombre de conditions doivent être réunies: (1) on doit pouvoir disposer d'éléments de preuve fiables, (2) les comportements attendus des composants doivent avoir été définis préalablement et (3) les parties doivent avoir précisé leurs intentions en matière de répartition des responsabilités. Dans cette thèse, nous apportons des éléments de réponse à ces questions en proposant un cadre formel pour spécifier et établir les responsabilités en cas de dysfonctionnement d'un logiciel. Ce cadre formel peut être utilisé par les parties dans la phase de rédaction du contrat et pour concevoir l'architecture de logs du système. Notre première contribution est une méthode permettant d'intégrer les définitions formelles de responsabilité et d'éléments de preuves dans le contrat juridique. Les éléments de preuves sont fournis par une architecture de logs dite “acceptable” qui dépend des types de griefs considérés par les parties. La seconde contribution importante est la définition d'une procédure incrémentale, qui est mise en œuvre dans l'outil LAPRO, pour l'analyse incrémentale de logs distribués.

# Contents

# Introduction

Software contracts usually include strong liability limitations and even exemptions of the providers for damages caused by their products. This situation does not favor the development of high quality software because software editors do not have sufficient economical incentives to apply stringent development and verification methods.

One of the main problems to define liabilities is that computer systems cannot be treated in the same way as physical systems and other traditional tangible goods. Because of their complexity it is often too hard to describe precisely all expected behaviors and potential defects that can occur during their execution. Another issue is the legal value of electronic evidence in court [Maurer 2004, Buskirk & Liu 2006, Insa 2006]. First, the production and manipulation of electronic evidence should follow specific rules which may depend on jurisdictions and types of trials. Another source of uncertainty is the fact that the weight of electronic evidence is not an absolute criterion and its final evaluation is left to the appraisal of the judge.

Taking up these challenges was precisely the main objective of the LISE[2] project. The work developed in this thesis, which has been carried out within LISE, is to define a formal framework which can be used in the elaboration a software contract to specify certain liabilities as precisely as possible.

Beyond liabilities, another key aspect of our work is the study of the use of the logs as digital evidence. The liabilities among the parties of a contract are characterized with respect to entries of the logs. We believe that this approach can not only help the contract elaboration process, but also improve the legal value of logs used as digital evidence.

## Structure of the Document

This thesis is structured in six chapters. In Chapter 1, we describe the context of the thesis and identify the main research challenges. We also provide a study of the main contributions in the state of the art. We conclude Chapter 1 with an introduction to the LISE project and its objective.

[2]LISE (Liability Issues for Software Engineering) was a project funded by ANR (Agent Nationale de la Recherche) under the SeSur 2007 programme (ANR-07-SECU-007). `http://licit.inrialpes.fr/lise/`

In Chapter 2, we start from the legal framework proposed by the lawyers of the LISE project and we derive a number of technical requirements for a framework for the specification of liabilities. This chapter also contains a review of the technologies used in the definition of our formal framework.

In Chapter 3, we introduce the first part of our framework dedicated to the representation and analysis of logs. This chapter also includes the specification of a log analysis tool capable of analyzing distributed logs. We also define an incremental version of this log analysis tool. In this chapter, we introduce a case study that is used to illustrate our approach throughout the document.

In Chapter 4, we define the second part of our framework dedicated to the specification of liabilities and we propose a systematic procedure to establish liabilities. In this chapter we also provide criteria to analyze log distributions in order to improve the legal value of the logs to be used as digital evidence.

In Chapter 5 we present the tool that we have implemented to establish liabilities and we introduce a language of properties that can be used to specify liabilities. In this chapter, we also discuss the performances of our tool and suggest some optimizations. Finally, we conclude this document with some perspectives.

# Chapter 1

# Context and State of the Art

In this chapter we describe the context of this thesis and we provide a study of the state of the art in related domains.

## 1.1 Context

Software systems have considerably grown in scale and functionality and this growth will inevitably continue in the future. Adequate functionality and quality for these systems is a crucial issue in a society that vitally depends on them. One must admit, however, that software systems are far from immune from failures. According to Charrette [Charette 2005], the most common factors of software failures are badly defined system requirements, the inability to handle software complexity and sloppy development practices. Charrette also points out that most software failures are predictable and avoidable but most organizations do not consider preventing failures as a critical issue.

Nevertheless, software failures may cause catastrophic losses of money, time or even physical damages [Birsch 2004]. For instance, in 2009, a component failure partially led to the derailment of a software controlled train in London [Branch 2010]. The problem occurred due to some conditions not considered in the signal control system which led the train controller to believe that the train was in a safe situation when it was actually not the case. In the health care sector, the malfunction of a software component during a LASIK[1] eye surgery could cause irreversible blindness [U.S. Food and Drug Administration 2011]. A wrong interpretation of a parameter to be provided to the system could lead to an over exposition of the laser permanently damaging the eye. In the financial sector, a human error (a trader accidentally mistyping the size of a trade) that should have been detected by the software led to almost a thousand points drop in the Dow Jones stock market [ABC 2010]. These examples show that software failures may have severe consequences.

[1] Laser-assisted in situ kerotomileusis is a surgery for correcting myopia, hypermetropia and astigmatism

One of the major goals of software engineering is thus to enable developers to construct systems that operate reliably despite their complexity. Software engineering methods greatly increase the understanding of software systems and can reveal inconsistencies, ambiguities and incompletenesses that might otherwise go undetected. However, despite all the efforts made during the development of software systems, software failures can still occur even in critical systems. Woodcock et all. point out that there is no way to guarantee that a complex system will operate without failures [Woodcock et al. 2009], mainly due to the fact that requirements are often difficult to elicit precisely. Another major reason is that software developers frequently cannot predict all the contexts in which their products will be used or integrated [Ryan 2003, Schneider 2009]. Last but not least, the priority granted by organizations to failures prevention is also a key factor in this context [Charette 2005].

## 1.2 Software Liabilities

Considering that failures may occur, the next question is how the liabilities for software errors can be defined and established. Some studies [Patel 2006, Patel 2007] suggest that very often liabilities are not defined very clearly in contracts between companies, although the precise specification of liabilities will more and more become critical to successful businesses. Other studies [Marotta-Wurgler 2007] show that software licenses usually include strong liability limitations or even exemptions of the providers for damages caused by their products. This situation does not favor the development of high quality software. In fact, experience shows that products tend to be of higher quality and more secure when the actors in position to influence their development are also the actors bearing the liabilities for their defects [Anderson & Moore 2009, Berry 2007, Ryan 2003].

The usual justification of software providers is the fact that software products are too complex and versatile objects whose expected features (and potential defects) cannot be characterized precisely, and which thus cannot be treated as traditional tangible goods [Schneider 2009, Ryan 2003, Birsch 2004]. Indeed, it is well known that defining in an unambiguous, comprehensive and understandable way the expected behavior of systems, integrating a variety of components, is quite a challenge. In addition, the establishment of a clear causality relationship between a failure of the system and the component that produced the error leading to the failure can also be a very complex task. Even when this relationship can be established, the liabilities may still depend on the precise specifications and commitments of the parties involved in the development of the product. For example, when a computer system fails due to one of its component, the producer of the component may be liable for the failure. However, it may also be the case that the system integrator is liable for using the component under certain conditions not assumed by the producer.

As a result of this complexity, when they are not specifically excluded, contractual liabilities are usually expressed in very general, or imprecise terms in software contracts [Patel 2006, Patel 2007]. Generally speaking, texts in natural language, even in simple

“legal language", often conceal ambiguities and misleading representations. This may cause a lower rate of compliant transactions resulting in potential financial penalties: “the average savings of transactions that are compliant with contracts is 22%” [Patel 2006, p. 1]. The situation is even worse when contracts refer to mechanisms which are as complex as software [Schneider 2009].

Another challenge concerning the liabilities for software systems is how to effectively establish them in case of incident. Usually, the investigations of legal disputes and crimes involving computer systems involve the use of digital forensics techniques. In [Richard III & Roussev 2006], Richard III and Roussev point out that the increasing complexity of computer system over the last years is demanding more attention from digital forensic investigators: digital forensics techniques need to take into account aspects such as how to extract relevant evidence to establish liabilities, how to efficiently analyze a large amount of data and how to automate the analysis process.

## 1.3 Research Issues

Specifying and establishing liabilities in case of litigation involving computer systems is generally a delicate matter. Taking up this challenge is precisely the objective of the work described here. Our starting point is the legal contract signed between the parties involved in the design or use of a computer system. We aim to propose a technical framework to define contractual liabilities in a precise and unambiguous way, to build evidence and establish such liabilities in case of failure. Obviously, any technical solutions or methodology in this context should comply with the legal requirements specially with respect to contract validity and evidence theory. The three main challenges to address this objective are the following:

1. **How to represent liabilities in a precisely unambiguous way?**

   The first challenge is to establish a precise relationship between the failures of the system and the parties liable for these failures. This requires a precise and unambiguous specification of the failures and possibly to use this specification to elaborate a valid liability agreement in a legal contract.

2. **How to produce the digital evidence to establish liabilities?**

   The second challenge is to ensure that convincing evidence will be available to establish liabilities in case of disagreement between the parties. This evidence should be sufficient (1) to show that a failure has effectively occurred, and (2) to identify all the incorrect behavior of the components which have led to the failure.

3. **How to establish liabilities in case of incident?**

   Once the digital evidence is collected, an analysis of this evidence is necessary to identify the liable party (or parties) according to the contract. Depending on the formalism used to define liabilities, this task can be more or less difficult and expensive. It is also necessary to define the amount of digital digital evidence necessary to establish liabilities and to deal with the additional complexity of distributed logs.

To address these issues, we follow an approach based on the *a priori* analysis of liabilities in order to define which evidence has to be produced to establish liabilities. In our approach evidence takes the form of *log files* containing information about the actions executed by the components of the system. We also assume the existence of a *contractual agreement* between parties. The contract describes the liabilities associated with a computer system including the content of the digital evidence and how they will be produced.

Several connected areas share part of our objectives and provide useful hints and results. We present three main areas related to the work described here, corresponding to the three challenges mentioned above:

- **Formal contract** formalisms share with our approach the objective to specify contracts as precisely as possible. These formalisms usually aim to provide mechanisms to express contractual obligations in a precise way and are closely related to our first objective. Contributions in this domain also include the analysis of contracts to detect inconsistencies, as well the use of contracts for the purpose of monitoring.

- **Digital evidence** frameworks share with our approach the objective to use digital information in a legal setting. Usually, the digital evidence used in forensic investigations should be in conformance with precise legal and technical requirements. Contributions in this domain include techniques to produce, check and analyze evidence that fulfill these requirements.

- **Trace analysis** proposals share with our approach the objective to analyze the behavior of computer systems based on their execution traces. Contributions in this domain provide helpful hints about the procedure that should be used to establish liabilities based on the observed behavior of the system.

In the following sections we sketch the main contributions related to our objectives in each of these areas.

## 1.4 Formal Contracts

Business contracts are usually considered as purely legal documents without strong connection with the day-to-day conduct of business of the companies. A better approach however

is to see the contract as a way to define more precisely the interactions between the parties and their responsibilities [Patel 2007]. For instance, besides expressing the obligations of each party, contracts may express the measures to be taken when a violation is detected [Molina-Jiménez et al. 2009]. In the past years, significant results have been achieved on the specification and analysis of contracts ([Giannikis & Daskalopulu 2011, Strano et al. 2009, Oren et al. 2008, Pace & Schneider 2009, Andersen et al. 2006] to cite a few).

Contractual clauses, which usually express obligations, permissions or prohibitions [Pace & Schneider 2009], can typically be expressed in deontic logic [von Wright 1951]. Deontic logic provides a very general framework with high expressive power but the price to pay for this expressiveness is the counter intuitive meaning of certain statements known as paradoxes [Meyer et al. 1994]. For example, the Ross's Paradox states that for any $X$ and $Y$ it is possible to show that $X$ is obligatory implies that $X$ or $Y$ is obligatory ($O(X) \Rightarrow O(X \vee Y)$). It seems odd, for example, that an obligation to "read a letter" implies an obligation "to read the letter or to destroy it" which can be interpret as the possibility to escape the obligation to read the letter.

To avoid these problems, other formalisms have been proposed to formalize contracts. In this subsection, we first identify the main requirements for a contract language. Then, we sketch the main formalisms in the literature and evaluate them with respect to these requirements.

### 1.4.1 Requirements for Contract Formalisms

The requirements for contract formalisms identified below are mostly based on [Pace & Schneider 2009, Patel 2007, Yao-Hua Tan 2001]. We divide these requirement into two types:

- **Expressiveness** requirements concern the type of statement that may be expressed in the formalism. They include:

    1. **Event-based vs. State-based** properties – some formalisms make it possible to express properties on states and other focus on events (or actions) properties.
    2. **Contrary-to-duty Obligations** – Contrary-to-duty obligations express a situation in which there is a primary obligation and a secondary obligation, which comes into effect when the primary obligation is violated. As an illustration, consider the sentence "There must be no failure. If there is a failure then it must be fixed within 3 days". The second sentence is a contrary-to-duty obligation because it is only considered if the first obligation is violated. A challenge in deontic logic and contract languages in general is the proper representation of contrary-to-duty obligations [Prakken & Sergot 1996].
    3. **Temporal Constraints** – One of the main features of contracts is the interaction between deontic and temporal modalities. Indeed, contractual obligations

or prohibitions usually come with a deadline which may be defined by fixed date, by a delay or by an event.

- **Analysis** requirements concern the analysis of contracts to detect inconsistencies, breaches and possibly the actions to be taken in case of breach. The main analysis requirements are:

  1. **Conflict Analysis** – A valuable goal is the analysis of inconsistencies between different clauses of the contract. For example, to detect that two obligations conflict with each other.
  2. **Compliance Analysis** – Another objective is to ensure for a given execution of the contract that the clauses of the contract have not been breached.
  3. **Blame assignment** – A more ambitious objective is to be able to assign the blame to a party (or several parties) in case of breach of the contract. Blame assignment mechanisms associate contractual violations with the parties of the contract that are liable for the violations.

In the following, we analyze the features of the main formalisms proposed in the literature to specify contracts and we assess them by the yardsticks of the above requirements.

### 1.4.2 The Contract Formalism by Jones et al.

In [Jones et al. 2003], the authors propose a formalism dedicated to the specification of financial contracts. The authors suggest that complex contracts may be formed by the combination of simpler contracts. They propose a set of combinators that are commonly used in contracts, such as *zero-coupon discount bound* used to express, for example, sentences like "receive $X$ on date $t$". Most of the statements that can be expressed using these combinators are related to deliver and payment commitments in the exchange of goods.

Jones et al. also introduce a combinator to express *limit clauses*. Limit clauses are sentences with a deadline or bound limit, e.g. "unless the temperature falls below zero" or "unless interest rates go above 6%". The combination of limit clauses with other types of combinators can be used to specify contrary-to-duty obligations.

Temporal constraints can also be expressed by combinators. Explicit time constraints may be represented with the combinator *when* which defines an obligation that should be activated when a given value is observed. Temporal constraints about the order of events can be represented using the combinator *anytime* which defines that a condition will eventually hold after a given value is observed.

Expressiveness requirements are summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| state-based | yes | yes | - focus on the exchange of goods<br>- contract between only two parties |

The proposed language involves a notion of *observable values*. Observable values are measurable quantities observed by the parties which can be used to specify conditions. For example, the temperature can be an observable value, and it is possible to use a combinator to express that "The client should pay $100 each month unless the temperature is below zero".

Although the authors do not provide specific details, they offer an implementation of the language using the Haskel language. This makes it possible to provide conformance checking mechanisms reading the set of observable values and ensuring that the clauses in a given contract are always respected.

In the proposed formalism, contracts are always bilateral: between the holder of the contract and a single counter-party. Therefore, there is no support for complex blame assignment mechanisms where multiple parties could be involved.

Analysis requirements are summarized in the following table:

| Conflict | Compliance | Blame Assignment |
|---|---|---|
| no | yes | no |

### 1.4.3 The Process-Oriented Event-Driven Transaction System (POETS)

In [Andersen et al. 2006], the authors extend the work of Jones et al. to encompass the exchange of money, goods and services between multiple parties. Andersen et al. propose a trace-based denotational semantics to specify contracts: a contract is formed by the combination of other contracts and consists of a set of traces. Each trace is a finite sequence of events that represents a way of concluding the contract successfully. The language is incorporated as a core component in a process-oriented event-driven transaction system (POETS) [Henglein et al. 2009]. A trace-based approach makes it possible to express statements concerning the expected occurrences of events.

POETS supports a variant of contrary-to-duty clauses using pairs of traces. The pair represents the choice between respecting or not the first commitment of a clause. However, this approach does not distinguish between the primary and secondary obligations. The trace-based semantics leads to a natural representation of temporal constraints w.r.t the order of occurrence of events. Explicit time constraints can also be represented based on deadlines. For example, the following obligation:

$$transmit(a_1, a_2, r, t)$$

means that the agent $a_1$ should transmit to the agent $a_2$ the resource $r$ before the time $t$ (deadline).

Expressiveness requirements are summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| event-based | yes | yes | - focus on the exchange of goods<br>- clauses with deadlines |

The authors also define a semantics for compliance checking based on event traces. The idea is to match the observed sequence of events with the expected events specified in the contract to check that no clause has been breached.

Analysis requirements are summarized in the following table:

| Conflict | Compliance | Blame Assignment |
|---|---|---|
| no | yes | no |

### 1.4.4 The Contract Language $\mathcal{CL}$

The contract language $\mathcal{CL}$ is introduced by Prisacariu and Schneider [Prisacariu & Schneider 2007]. Like POETS, $\mathcal{CL}$ relies on a trace-based semantics. $\mathcal{CL}$ introduces deontic modalities to provide the notions of obligation and permission. Typically, a deontic logic uses the operator $Ox$ to state that "it is obligatory that $x$", the operator $Px$ to state that "$x$ is permitted" and the operator $Fx$ to state that "$x$ is forbidden". In $\mathcal{CL}$, a contract defines a set of actions and the deontic operators are used to define assertions about these sets of actions. As an illustration, let us consider a simple contract stating that a provider must deliver a product to a receiver (*deliver_product*) and, upon the service delivery, the receiver must pay for the product (*pay_product*). This contract can be specified in $\mathcal{CL}$ as follows:

$$O(deliver\_product) \wedge$$
$$[deliver\_product]O(pay\_product)$$

The syntax of $\mathcal{CL}$ allows the specification of action sequencing ($a_1 a_2$), alternative ($a_1 + a_2$) and concurrency ($a_1 \& a_2$).

In $\mathcal{CL}$, deontic operators are applied to actions rather than states. The authors justify this choice by the observation that contracts usually describe what may or may not be performed, rather than what may or may not be the state of affairs. In consequence, a statement such as "the bandwidth should be more than 20kbps" cannot be expressed directly in $\mathcal{CL}$, because it defines an obligation on a state rather that an action.

$\mathcal{CL}$ supports contrary-to-duty obligations through the use of an operator representing the lack of an action (noted $\overline{a_1}$). For example the statement $O(a_1) \wedge [\overline{a_1}]a_2$ expresses that $a_1$ is an obligation, and in the absence of $a_1$ then $a_2$ becomes an obligation.

Temporal constraints are expressed in $\mathcal{CL}$ through the application of temporal connectives to obligations or permissions. The language provides the connectives until ($\mathcal{U}$) and next ($\bigcirc$) of standard temporal logic, which make it possible to express constraints on the order of actions but without support for explicit time constraints. It is possible however, to write liveness properties (e.g. "provider must deliver eventually") using the connective $\mathcal{U}$.

Expressiveness requirements are summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| event-based | yes | yes (non timed events) | - deontic logic modalities |

In [Kyas et al. 2008], the authors propose a method to generate monitoring mechanisms for contracts written in $\mathcal{CL}$. These mechanisms take the form of automata which accept only the traces that respect the clauses of the contract and can be used to check the compliance for a given execution of the contract.

In [Fenech et al. 2009a] Fenech et al. define a formal notion of conflict in a contract. Intuitively, a conflict can be detected when an action is both imposed and forbidden; or both permitted and forbidden; or when two contradictory actions (noted $a_1 \# a_2$) are permitted. The authors describe a procedure to detect conflicts by constructing the set of automata representing the contract and searching for conflicts in the traces accepted by these automata.

Analysis requirements are summarized in the following table:

| Conflict | Compliance | Blame Assignment |
|---|---|---|
| yes | yes | no |

Hvitved [Hvitved 2010] proposes CSL, a language to specify contracts with a trace-based semantics. CSL is very similar to the formalism proposed by Andersen et al. [Andersen et al. 2006] and Pace and Schneider [Pace & Schneider 2009] and provides the same expressiveness. The author argues that although contract formalisms are more targeted towards the elaboration of the contract itself, *blame assignment* is a fundamental requirement for contract languages. An interesting aspect of the language is the fact that the clauses of the contract are associated with the parties that should be held accountable in case of violation. CSL provides an abstract definition of a run-time monitoring mechanism that receives as input a trace and return "yes" if any violation is detected and, in case of violation, the parties liable for the violation.

The analysis requirements for CSL can be summarized in the following table:

| Conflict Analysis | Compliance | Blame Assignment |
|---|---|---|
| yes | yes | yes |

### 1.4.5 The Rule-Based Contract Language RuleML

RuleML [RuleML 2011, Governatori 2005] is a XML-based contract language which uses the notion of *rules* to express obligations. Rules are statements of the form event-condition-action expressing that when a given event takes place, a given action must occur under a given condition. RuleML is "semantically neutral" language, meaning that there is no semantics attached to it.

To support contrary-to-duty clauses, RuleML introduces the connective $\otimes$ used to state reparation for clause violations. For example, $Ox \otimes Oy$ is read as "$Oy$ is the reparation of the violation of $Ox$". This means that $x$ is obligatory, but if the obligation $Ox$ is not fulfilled then the obligation $Oy$ becomes active.

In RuleML, it is possible to express temporal constraints using conditions attached to the rules. One can specify, for example, that "the client must login before making a reservation" or "the provider must deliver the service 5 days after a request". Since RuleML is semantically neutral, many frameworks propose implementations of RuleML (e.g. jDREW [jDREW 2011] is a Java-based deductive engine for RuleML) providing different means to express temporal constraints.

Expressiveness requirements are summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| state and event based | yes | yes (no semantics) | -rule based syntax<br>- no fixed semantics<br>- different implementations |

DR-CONTRACT [Governatori & Pham 2009] extends RuleML to include compliance checking. The authors propose an inference engine that receives a list of observed events and finds if these events violate any rule in the contract. Another approach, propose by [Blom et al. 2004], consists of translating each rule into an automaton and tracking rule violations based on the execution trace.

In [Governatori & Pham 2009], the authors introduce a Defeasible Deontic Logic Violations into RuleML based on a hierarchy relationship between rules which express the fact that some rules may overrule other rules. For example, consider the two rules:

$r_1$: The price of the service for all clients is 300$
$r_2$: Premium clients get a 5% discount

We can solve this conflict stating that $r_2$ has a higher priority then $r_1$. This logic can be used to detect and solve conflicts in the specification of rules.

Analysis requirements are summarized in the following table:

| Conflict Analysis | Compliance | Blame Assignment |
|---|---|---|
| yes | yes | no |

### 1.4.6 The IST Contract Project

The IST Contract Project [IST 2011] is a research project funded by the European Commission that aims to cover both theoretical and practical aspects of the specification of electronic business-to-business contracts. The project proposes a formalism [Oren et al. 2008] to specify contracts based on the notion of *normative statement* (or simply *norm*). A contract is defined as a set of norms that describe the obligations in the contract.

A norm consists of five components: the norm type, an activation and an expiration condition, a goal and a target. The norm type corresponds to a deontic modality (obligation, permission or prohibition). The activation and expiration conditions define respectively when the norm should become activated and deactivated. The goal defines the actions that may/must be performed if the norm is activated. Finally, the target describes the agent(s) to whom the norm applies.

Except for the expiration condition, norms are very similar to the event-condition-action approach. The authors propose an operational semantics for this language, based on the notion of *normative states*. Normative states intuitively divide norms into a set of active norms, a set of inactive norms and a set of expired norms according to the activation and expiration conditions. The three classes of normative states are updated according to the actions that are performed.

Contrary-to-duty obligations may be expressed using the activation and expiration conditions to specify that when the primary obligation is expired then the penalty obligation should be activated. The authors describe the possibility to use activation and expiration conditions to express temporal constraints. However, the semantics of the language does not involve any notion of time.

The expressiveness requirements for this formalism can be summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| event-based | yes | yes (not formalized) | - rule-based syntax and semantics |

Although the authors do not propose monitoring mechanisms, they suggest to use an event calculus implementation to keep track of normative states [Farrell et al. 2005]. They also refer to [Daskalopulu 2001] to describe how Petri nets could be used to perform contract monitoring.
The analysis requirements are summarized in the following table:

| Conflict Analysis | Compliance | Blame Assignment |
|---|---|---|
| no | yes | no |

### 1.4.7 The Contract Formalism of Xu and Jeusfeld

Xu and Jeusfeld [Xu & Jeusfeld 2003] propose a formalism to specify contracts based on *commitment graphs*. In a commitment graph, the nodes represent the parties and the edges represent the actions that can be performed. Contractual commitments are defined as sequences of actions. Each action specifies a set of expected inputs and outputs. For example, if a provider delivers some goods to a client (action *delivery*), the client should pay the costs of the goods (action *payment*); the output of *delivery* is a condition to activate the commitments concerning *payment*.

Temporal commitments can be expressed only as constraints on the order of execution of the actions and there is no support for explicit time constraints. There is no mention of support to contrary-to-duty clauses. The expressiveness requirements for this formalisms can be summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| event-based | no | yes (non timed events) | - input/output activate obligations |

An interesting aspect of this approach is that monitoring mechanisms not only detect violations of the contract, but also determine the parties responsible for these violations. In [Xu et al. 2005] the authors propose a model that relates a contractual commitment with the parties that should be responsible for the violation of this commitment. Through the analysis of execution traces it is possible to find which expected actions and outputs are missing, then detect when a violation occurred and what are the parties liable for the violation.
The analysis requirements can be summarized in the following table:

| Conflict Analysis | Compliance | Blame Assignment |
|---|---|---|
| no | yes | yes |

### 1.4.8 Contracts for Services

Contracts can also settle the terms of a service rather than a product. In this case, it is common to use a Service Level Agreement (SLA). A SLA is the part of a service contract that formally defines the conditions of the service. For example, Internet Service Providers (ISP) may include a SLA within the terms of their contracts with customers, specifying certain aspects of the service, such as the maximum time of recovery from failures or the average connection speed.

During the last decade, many languages have been proposed to specify SLAs. SLA languages usually provide domain-specific support for defining reliability, latency or throughput constraints for services. For example, Web-Service Level Agreement language (WSLA) is a language developed by IBM [Keller & Ludwig 2003] for the specification of web-services.

WSLA provides a set of *service level parameters* that specify which quantities should be measured, how each quantity should be measured, who is responsible for monitoring it and where the measurement can be retrieved. Constraints over the measured quantities can be expressed using pre-defined functions and predicates that can be combined hierarchically.

The contract formalisms described in the previous sections can also be extended to specify SLAs. For example, RBSLA [Paschke 2005] is a framework that extends RuleML to include SLA-specific elements such as metrics and domain-specific vocabularies. RBSLA supports a flexible syntax to describe rules concerning the conduct of services in general and provides an intuitive way to express contractual constraints for services.

Contrary-to-duty obligations are commonly supported by SLA languages. In SLAs, contrary-to-duty obligations usually specify penalties for the parties who do not satisfy their obligations.

Most SLA languages can express temporal constraints with an explicit notion of time. For example, SLAs often include clauses stating the maximum amount of time the service provider can take to answer a request from the client.

The expressiveness requirements for SLA formalisms are summarized in the following table:

| Properties | Contrary-to-duty | Temporal constraints | Specific features |
|---|---|---|---|
| - mainly state-based | yes | yes | - focus on the quality of service<br>- domain specific frameworks |

Run-time mechanisms for monitoring SLAs is one of the main features of these languages. Some approaches go a step further, such as [Skene et al. 2007] which defines the *monitorability* criterion to classify SLAs. The monitorability is defined according to a trust relationship between the parties. For example, a client may trust the Internet provider to observe the actions of the service and inform if the clauses of the contract have been violated or not. The authors propose a procedure to build SLAs with acceptable levels of monitorability.

Some SLA formalisms are also endowed with mechanisms for conflict detection. The RBSLA framework includes a method to detect conflicts in SLA and to automatically avoid or resolve conflicts [Paschke & Bichler 2005]. For example, an authorization conflict where a clause forbids and another clause allows at the same moment the execution of a request action may be solved by rejecting the request actions until the conflict is solved.
The analysis requirements are summarized in the following table:

| Conflict Analysis | Compliance | Blame Assignment |
|---|---|---|
| yes | yes | no |

### 1.4.9 Summary

Regarding the expressiveness requirements we can conclude from this study that most existing contract formalisms are able to represent contractual liabilities providing different programming paradigms and various degrees of expressiveness. Clearly, a lower expressive power can be a design choice rather than a weakness because trade-offs between expressiveness and simplicity are necessary to make the formalism usable.

With respect to the analysis requirements, we can observe that only CLS and; Xu and Jeusfeld consider blame assignment. In addition, these approaches only consider deterministic blame assignments. This means that they are limited to contracts where violations can always be uniquely assigned to a certain set of parties in the contract. Typically, traces that reveal several errors from different components cannot be analyzed within these frameworks. Both approaches define liabilities by associating clauses of the contract with the parties that should be held accountable for the breach of the clause.

Another conclusion of this study is that, besides SLAng [Skene et al. 2007], no formalism addresses the issue of whether or not the digital evidence used to prove a contract violation can be trusted. Usually, these approaches assume that the digital evidence is trustworthy because it is secured by other means, such as specific tamper-proof hardware or security measures. We provide a study of the techniques of digital evidence in the following section.

## 1.5 Digital Evidence

Digital forensics involves the investigation of material found in digital devices, namely *digital evidence*, generally with the aim to use such material in legal procedures. Usually, digital forensics investigations concerns security attacks and computer crimes investigations [Carrier 2003, Reith et al. 2002]. Therefore, the works on digital evidence usually do not assume the existence of a legal contract. Nevertheless, this domain shares part of our objectives to analyze digital information in a legal setting, in particular how to manage logs as digital evidence in a legal context. In this section we explore the main research contributions of the use of digital data as evidence.

### 1.5.1 Legal Evidence Requirements

From a legal point of view, an evidence (digital or nor) is a proof legally presented at a trial which is intended to convince someone of alleged facts[2]. Legally speaking, three aspects have to be considered to assess the legal value of evidence:

- Admissibility in court: to be admissible in court, the evidence must comply with specific legal rules that may depend of the type of its nature (physical, digital, testimony,

[2]The People's Law Dictionary available at: http://dictionary.law.com

genetic, etc.).

- Relevance: the evidence must prove to be relevant and have a significant relationship with the case in question, i.e. evidence must have a significant role to prove something important in a trial.

- Probative value: the strength the evidence is associated with its capacity to convince or persuade a judge or a jury of a certain fact.

The admissibility in court may depend on jurisdictions and types of trials. For example, in most jurisdictions a communication recorded without the knowing of the participants cannot be used against them unless the recording has been authorized by a court. Another potential obstacle to the use of log files in court could be the principle according to which "no one can form for himself his own evidence". It seems more and more admitted however, that this general principle allows exceptions for evidence produced by computers. As an illustration, the printed list of an airline company showing the late arrival of a traveller at the boarding desk was accepted as evidence by the French "Cour de cassation"[3].

The relevance of digital evidence depends on the causality relationship which can be established between the facts. This issue is related to the analysis of the digital evidence (Section 1.6). The probative value of a piece of evidence is not an absolute criterion and its final evaluation is left to the appraisal of the judge and/or jury. However, there are desirable characteristics which are likely to increase the probative value of digital evidence. For example, specific protocols may provide guarantees of authenticity of data stored in the logs.

In the following section, we review the security requirements of the logs that may increase their probative value as digital evidence.

### 1.5.2 Log Security Requirements

Logs are records of observable *events* performed by a computer system. Logs are composed of *log entries*, and each log entry contains information related to a specific event that has occurred within a system.

We are not aware of specific regulations dedicated to the use of logs as digital evidence. However, standard organizations, such as the National Institute of Standards and Technology (NIST), propose a list of measures for the logs management. For instance, the "Guide for Computer Security Log Management" [Kent & Souppaya 2006] contains important requirements and goals that should be taken into account to establish a policy to generate, build, store and analyze the logs of a system, such as follows:

- What components of the system should be logged.

[3] *Cass. civ.* $1^{st}$*, July* $13^{th}$ *2004: Bull. civ. 2004, I,* $n^o$ *207*

- What data should be logged for each type of event.
- What protocol should be used to transmit the data to build the log file.
- How confidentiality, integrity and availability of the information in the logs should be protected either during transmission or in memory.

Some of these aspects are supported by the logging protocols used to produce the logs. Logging protocols are technical solutions dedicated to the transmission and storage of log data. They may describe the structure of a log file and how the information of each log entry is stored into the file. A logging protocol may also specify the communication protocol that should be used to transmit the data of the logs.

Kenneally [Kenneally 2004] sketches several considerations to take into account for logs to be admitted as digital evidence and relates them to the guarantees of "trustworthiness" of the evidence. Accorsi [Accorsi 2009] analyze security requirements of logging protocols such as integrity and authenticity. We describe here some of most important requirements identified in these works:

1. **Authentication** – the entries in the logs must come from authorized and identifiable devices. This means that it is always possible to identify the components involved in the event represented by each log entry. Besides this we can expect that once a log entry is produced is not possible to repudiate the observed event.

2. **Integrity** – the events must not be modified during transmission and the logs cannot be tampered once recorded. In particular it must be impossible for an attacker to tamper the information of the log by adding, removing or changing the content of log entries either during their transmission or within the log files.

3. **Confidentiality** – it may be required that some information contained in the logs remains confidential during transmission and storage. This requirement is usually address through the use of cryptographic techniques before transmitting or storing the log data.

In order to make these requirements more precise, we introduce three types of entities based on the model proposed in [Accorsi 2009]. *Sensors* are software components that observe the events and automatically generate the corresponding log entries. These log entries are transmitted to a software called the *collector* that is responsible for adding the log entry into a log file. Finally, an *investigator* is an entity that can check the integrity of the log files.

Authentication is usually achieved using algorithms based on digital signature, for example, using asymmetric keys. Each entry in the log is digitally signed by a sensor. This provides guarantees that the log data is produced by authorized devices and once an authenticated entry is registered in the log it cannot be repudiated.

Confidentiality in the transmission is established by the communication protocol used to transmit the log data between the sensor and the collector. During the storage the confidentiality is ensured by the application of cryptographic techniques (either symmetric or asymmetric).

Integrity is a more complex aspect because different types of attacks have to be addressed. For example, a logging protocol may protect integrity of the logs against an attacker adding entries into the log but not against an attacker deleting entries. Usually logging protocols specify a list of possible threats such as the access to the cryptographic key of a sensor. Just like confidentiality, integrity has to be ensured within the process of transmission of log data between the sensor and the collector and when the data is stored into the collector memory. In addition to cryptographic techniques, security mechanisms such as access control to log files can be used.

In the following, we analyze the features of the main logging protocols proposed in the literature and we assess them by the yardsticks of the three security requirements given above. Since authentication and confidentiality are usually addressed with standard solutions, we mainly focus on integrity.

### 1.5.3 The Syslog Standard

The Syslog standard [Gerhards 2001] specifies a format of log entries including a type (such as `mail` or `news`) and a priority (such as `emergency` or `warning`). To transmit log data, Syslog uses the User Datagram Protocol (UDP). In Syslog, log entries are stored in cleartext and are not authenticated. Syslog was initially not intended to be a secured logging protocol. However, in the past years various extensions of the traditional Syslog standard have been proposed, improving data transmission security. Here, we present three of these extensions and describe their security features.

Syslog-ng [BalaBit IT Security 2011] and Reliable Syslog [New & Rose 2001] both extend Syslog to provide reliable transmission using the Transmission Control Protocol (TCP). Syslog-ng supports confidentiality during transmission of data using the Transport Layer Security (TLS) protocol. Reliable Syslog provides mechanisms for authentication and for the protection of integrity of entries during transmission, using the Blocks Extensible Exchange Protocol (BEEP) [Rose 2001].

Syslog-sign [Kelsey et al. 2009] also extends Syslog to provide authentication. It also provides support to log integrity checking against entry modification and deletion during transmission and storage. These requirements are achieved using the notion of *signature blocks* (Figure 1.1). The idea is to compute a hash of all combined previous entries to produce a digital signature (using SHA-1 and DSA) of the log file. This stamp is computed and safely stored periodically or after the insertion of a new entry. To verify the integrity of a log file against missing entries or tampered entries one has to compare the signature block of the log content with the signature block stored.

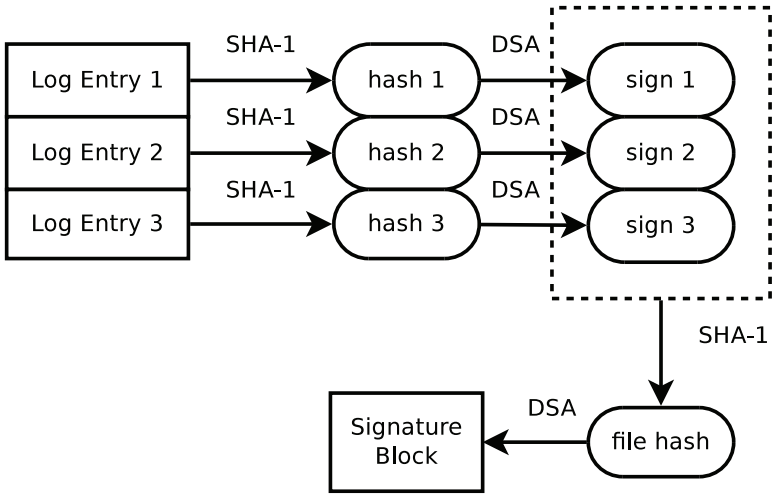



Figure 1.1: Signature block creation

The following table summarizes how Syslog and its extensions fulfill the three security requirements:

| Protocol | Authentication | Integrity | Confidentiality |
|---|---|---|---|
| Syslog | no | no | no |
| Syslog-ng | no | transmission | yes |
| Reliable Syslog | yes | transmission | no |
| Syslog-sign | yes | storage/transmission | no |

### 1.5.4 The Schneier and Kelsey Logging Protocol

Schneier and Kelsey propose in [Schneier & Kelsey 1999] a protocol for secure logging mainly focused on the protection of the data stored in memory (as opposed to transmission). The main techniques used to secure the logs are *hash chains* and *evolving cryptographic keys*. A hash chain is a successive application of a cryptographic hash function to a string. For example, let $h$ be a hash function and $s$ a string, then $[h(s), h(h(s))$ and $h(h(h(s)))]$ is a hash chain of length 3, often denoted $h^3(s)$. Evolving cryptographic keys consists of systematically changing the cryptographic key over time, possibly as a function of the previous keys, with the purpose of limiting the damage of attackers who obtain a key. Based on these two techniques the authors provide algorithms to create logs and append authenticated entries to them.

Schneier and Kelsey propose a log entry format consisting of four parts as follows:

$$L_i = \boxed{M_i} \boxed{\{E_i\}_{K_i}} \boxed{H_i} \boxed{C_i}$$

1. The *authorization mask* $M_i$ controls the access to the contents of the entry, i.e., only investigators authorized in $M_i$ gain access to the contents of the entry.

2. The value $E_i$ of the entry encrypted, with a key $K_i$. This key is generated through the application of a hash function to a combination of the authorization mask $M_i$ and an *authentication key* $A_i$ possessed by the sensor. Once generating $K_i$, a new authentication key $A_{i+1}$ is automatically produced.

3. The $H_i$ hash chain value is generated from the hash of $M_i$, $\{E_i\}_{K_i}$ and the hash chain value associated to the previous entry $H_{i-1}$. Any change in an entry can be detected as an error in the hash chain, which ensure message integrity.

4. $C_i$ is the *authentication code* computed using the authentication key $A_i$ of the sensor that generates the entry.

An informal threat analysis of the Schneier and Kelsey protocol revealed attacks in which modifications cannot be detected [Stathopoulos et al. 2006, Holt 2006, Accorsi 2006]. For example, an attacker can truncate a log file, without breaking the hash chain. To address this weakness, several evolutions of this protocol have been proposed.

In [Stathopoulos et al. 2006], the authors show that internal attacks allow an attacker knowing a given authentication key to reconstruct parts of the log in a way that tampering could not be detected. To solve this problem the authors introduce a "regulatory authority" which ensures that the log system follows the protocol. The idea is similar to the signature blocks used in the Syslog-sing protocol (Section 1.5.3). The regulatory authority periodically generates signature blocks for logs. In case of suspicious actions. the current signature block of the log is compared with the signature block stored by the regulatory authority.

In [Holt 2006], the authors point out a weakness related to the use of symmetric keys: an investigator should possess the key used to authenticate log entries and has the ability to tamper log entries. The authors propose Logcrypt, a protocol based on public key cryptography to compute the authentication code of the message. An asymmetric key approach makes it possible the use of two keys for verification and authentication respectively.

In [Sackmann et al. 2006, Accorsi 2006], the authors modify the Schneier and Kelsey protocol to take into account authentication during the storage and the transmission of the log entries. They introduce a public key infrastructure where each message is encrypted and signed before transmission to the collector. The proposed protocol also ensures message integrity during transmission using timestamps. For every new log entry received by the collector, it sends back to the sensor an acknowledgment message with the $C$-value of the entry and a timestamp. This message is stored by the sensor and an investigator can use this value to verify the integrity during the transmission of log entries. To avoid the possibility of attacks truncating log files, the authors propose that instead of authenticating each entry $L_i$ based on the content $E_i$, the entry should be authenticated based on the hash chain value $Y_i$.

The following table summarizes how the above logging protocols fulfill the security requirements:

| Protocol | Authentication | Integrity | Confidentiality |
|---|---|---|---|
| Schneier and Kelsey | yes | storage | yes |
| Stathopoulus et al. | yes | storage | yes |
| Logcrypt | yes | storage | yes |
| Sackmann et al | yes | storage/transmission | yes |

### 1.5.5 The Ma and Tsudik Logging Protocol

Ma and Tsudit [Ma & Tsudik 2009] propose a logging protocol ensuring integrity without any auxiliary information associated to the entries. In this protocol a log file consists of two parts: the sequence of entries $L_1, L_2, L_3, \ldots, L_i$ and two codes $V_i$ and $T_i$ corresponding to the last entry $i$. The structure of a log file is defined as follows:

| $L_1, L_2, L_3, \ldots, L_i$ | $V_i$ | $T_i$ |
|---|---|---|

In addition, the collector possesses two symmetric keys $A_i$ and $B_i$ which are automatically updated after each new entry. The procedure to add a new entry $L_{i+1}$ is:

1. Computation of two digital signatures $Sign_{A_i}$ and $Sign_{B_i}$ of the entry $L_{i+1}$ using respectively $A_i$ and $B_i$.

2. Computation of the codes $V_{i+1}$ and $T_{i+1}$ such that:
   $V_{i+1} = h(V_i + Sign_{A_i})$
   $T_{i+1} = h(T_i + Sign_{B_i})$
   where $h$ is a hash function.

The initial keys ($A_1$ and $B_1$) are randomly build before the first entry and stored both by the sensor and by the collector.

The idea behind the two codes $V_i$ and $T_i$ is that, at any time, an investigator entity can check the integrity of the log by obtaining the key $A_1$ and computing $V_i$ incrementally from the log content. The investigator not knowing the key $B_1$, cannot add valid entries in the log file.

Considering that log files may be very large, the first benefit of this approach is that the collector stores only two codes and two keys for each log file. The second benefit is that investigators cannot acquire the information necessary to change the log files.

The following table summarizes how the logging protocol proposed by Ma and Tsudit fulfill the security requirements:

| Protocol | Authentication | Integrity | Confidentiality |
|---|---|---|---|
| Ma and Tsudit | yes | storage/transmission | no |

### 1.5.6 Searching Information in Encrypted Files

Most of the logging protocols presented so far focused on the integrity of the log entries. To ensure confidentiality, some logging protocols involve the encryption of all the content of the log file. The downside of this approach is that encrypted files are less convenient for information extraction, which is a requested feature for the analysis of digital evidence. A naive solution would be to decrypt all the file in before starting the analysis. This approach may involve a high computational cost and unintended access to classified data by an investigator. To solve this problem some authors propose a secure storage solution for encrypted log files to be analyzed in an efficient way. In the sequel, we focus on two of these approaches.

In the protocol proposed in [Waters et al. 2004], each entry $L_i$ of the log is defined as follows:

$$L_i = \boxed{\{E_i\}_K \mid H_i \mid c_{w_1}, c_{w_2}, c_{w_3}, \ldots, c_{w_n}}$$

- The content $E_i$ encrypted with a key $K$.
- A part of the hash chain $H_i$.
- A set of codes $c_{w_1}, c_{w_2}, c_{w_3}, \ldots, c_{w_n}$ called *keyword information* used to search the entries.

To add a new log entry, the collector first automatically extracts the keywords $w_1, w_2, w_3, \ldots, w_n$ from the entry based on the structure of the log entries. The procedure to extract the keywords depends of the context of the application. As an example, the authors provide a log file including the queries sent from a client to a database and suggest to use as keywords the names of the tables and columns used in the queries.

After extracting each keyword, the collector computes the keyword information through the application of a series of hash encryption function using the key $K$ and the keyword itself. Finally, the collector writes the entry with the encrypted content, the part of the hash chain and the list of keyword information.

To search for an entry in the log, the investigator sends to the collector a query containing a keyword $w$. The collector then computes $c_w$ and searches for matching log entries. Log confidentiality is preserved because the investigator entity does not know the key $K$ and therefore cannot see the information in $\{E\}_K$. The authors also propose a similar schema using asymmetric cryptography that supports authentication.

In [Ohtaki 2008], Ohtaki proposes a solution based on Bloom filters. A Bloom filter (Figure 1.2) is a probabilistic data structure used to test whether an element is a member of a set. Filters are represented by an array of $m$ bits and a set of $r$ independent hash functions $h_1, \ldots, h_r$. They propose to incrementally construct for each log file a filter which contains information about the entries of the log. To do so, we start with all $m$ bits set

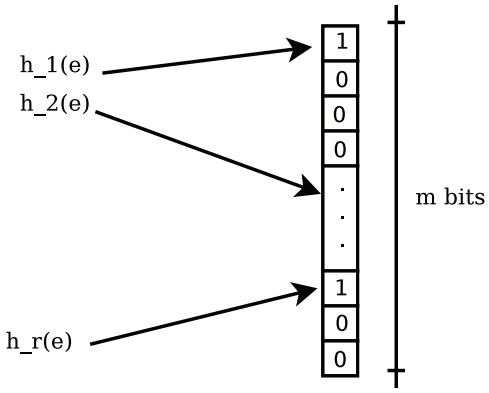



Figure 1.2: Bloom Filter Structure

to 0. To add an entry $e$ in the log, we set the bits $h_1(e), \ldots, h_r(e)$ to 1. This procedure is repeated for each entry of the log.

To decide if an entry $x$ belongs to the log, we check in its filter if either one of the bits $h_1(x), \ldots, h_r(x)$ is equal to 0, which means that $x$ is not an element of the log. If all bits $h_i(x)$ are equal to 1 then $x$ is a member of the log. Using Bloom filters can lead to false positives (conclude that an element belongs to the log when in fact it does not) because the bits produced by $x$ in the hash functions may have been produced by others entries. However, there is no probability of false negatives (that an element does not belong to the log when in fact it does). The rate of false positives can be managed by tunning the size of the values $m$ and $r$. Low values for $m$ and $r$ can lead to a filter fulfill with 1's for logs with a large number of entries. The higher the values of $m$ and $r$ the lower the probability of false positives. In this case, every query results that the entry belongs to the log.

The following table summarizes how the above solutions fulfill the security requirements:

| Protocol | Authentication | Integrity | Confidentiality |
|---|---|---|---|
| Waters et al. | yes | storage | yes |
| Ohtaki | no | storage | yes |

### 1.5.7 Summary

From the above study we may conclude that a number of cryptographic solutions have been proposed to ensure the main log security requirements (authentication, integrity and confidentiality). As pointed out in [Accorsi 2009, Kent & Souppaya 2006], besides cryptographic techniques, other security measures must be put in place to ensure the security of the log files. For example, the access control to the log files should be granted only for authorized entities. Further guarantees should be provided with respect to the whole log management process [Kent & Souppaya 2006]:

1. *Log Monitoring* – one must ensure that the information registered in the logs corresponds to the real events.

2. *Log Analysis* – log files should be analyzed according trusted methods and tools. This issue is studied in more details in the following section.

To ensure the strength of the log based evidence, it is recommended to define precisely all the technical steps for the production of the log files, their storage and the means used to ensure their authenticity and integrity.

## 1.6 Trace Analysis

Trace analysis consists in using verification techniques to check whether the execution of a computer system (represented by its traces) satisfies or violates a given property [Leucker & Schallhart 2009]. Trace analysis has been applied in various domains including model checking, runtime verification and diagnosis. In this section, we sketch the main results on trace analysis related to this thesis. We use the term *trace* here to represent a set of observations about the execution of a computer system, the term *event* to represent the entries in a trace, and the term *parameters* to represent the information in the events. A trace usually defines an order relation between its events (or a subset of its events) that correspond to the chronological order. Some authors, e.g. [Barringer et al. 2010a], use the term "*log*" to refer to the actual files containing the record of the events, and the term "*trace*" to represent an abstraction of the logs. In this section, we chose to adopt a uniform terminology and use the single term "*trace*".

### 1.6.1 Challenges in Trace Analysis

The main challenge of trace analysis is to provide a way to verify a given property of a given trace (or set of traces). Works related to trace analysis usually can be classified according to three main parameters:

1. The structure of traces.

2. The expressiveness of the properties.

3. The verification algorithm.

The first aspect is relative to the content and organization of events in the traces, i.e., the distribution of the information among the traces. For example, some authors consider a single trace representing the complete execution of the system (e.g. [Barringer et al. 2010b, Bauer et al. 2006]), when others propose models to represent and analyze distributed traces where the events can occur concurrently (e.g. [Arasteh et al. 2007, Hallal et al. 2006]).

The second aspect is relative to the expressiveness of the language used to state properties (temporal conditions, causality conditions, deadline, etc.). For example, properties for reactive systems (with potential infinite executions) are commonly specified using Linear Temporal Logic (LTL) (Section 1.6.3). However, this logic is not expressive enough to state properties of real-time systems with time constraints, that are often specified using an extension of LTL called Timed LTL.

Finally, the third aspect is relative to the decidability and complexity of the verification. This aspect is related to the previous ones because the complexity of the verification usually depends on the richness of the types of traces and the expressiveness of the language of properties.

In the following, we identify requirements associated with each of with these aspects that allow us to characterize trace analysis proposals according characteristics of our interest. These requirements will then be used to analyze existing solutions in a systematic way.

### 1.6.2 Requirements for Trace Analysis

We consider successively the requirements associated with each of the aspects mentioned in the previous section:

- **Structure** requirement concerns the way the trace are represented. The main categories of structures are *distributed* or *centralized*. In the first case, a total ordering between events can be assumed, which is not true for the second case.
- Expressiveness requirements include the following:
  - **Parametric properties** – it is possible to generalize properties for a given set of events and parameters [Chen & Roşu 2009]. As an illustration of parametric property, consider a trace that contains the communications between a supplier and its costumers. One can specify a parametric property to verify if a given costumer (defined by its ID) sends a request to the supplier. The parametric property can then be instantiated for any costumer using his ID.
  - **Sub-trace verification** – it is possible to specify the specific parts of the traces on which a property should be verified. For example, one may state that a property to check if "component $A$ has sent a request" should be evaluated using only the traces of $A$. This approach makes it possible to reduce the amount of trace in the verification and to analyze larger logs at a reasonable price.
- Algorithm requirements concern the way that a given property is evaluated. They include support to:
  - **Timed verification** – it is possible to bound the period of time during which a property is supposed to hold (and is verified). Typically, this kind of property involves a notion of deadline.

– **Online/offline verification** – the verification algorithm may be performed *online* or *offline* [Leucker & Schallhart 2009]. Online verification is commonly used in situations where a quick response is necessary, for instance, in intrusion detection. Offline verification is preferred when time is not of the essence, for instance in diagnosis. In general, solutions that support online verification take into account *future incompleteness* in traces because future events may change the value of a property.

A large part of the body of work on trace analysis refers to LTL to state properties over traces. In the following section we provide a brief review of LTL and its application to express properties over traces. Finally, we summarize the features of the main trace analysis solutions proposed in the literature and we assess them by the yardsticks of the above set of requirements.

### 1.6.3 LTL Review

Linear Temporal Logic (LTL) is a logic introduced by Pnueli [Pnueli 1977] that allows reasoning about temporal conditions over sequences. For example, in LTL it is possible to expressing that "some event occurs in the future" or "some event does not occur until another event occur". LTL was initially proposed to describe the behavior of systems over infinite sequences of states in a computer system, but it can also be used to verify properties over finite sequences of events in a trace [Bollig & Leucker 2003, Bauer et al. 2011].

LTL extends propositional logic including the temporal operators *next, eventually, globally* and *until*. These operators are defined as follows:

Let $\phi$ and $\psi$ be properties in propositional logic and $T$ a trace. We note $T \models \phi$ to state that $\phi$ holds for $T$. Let $T_n$ be the suffix trace obtained from the $n$-th event of $T$ assuming zero-based index, i.e. $T_0 = T$. We have:

| Operator | Notation | Semantics | Interpretation |
|---|---|---|---|
| Next | $\bigcirc$ | $T \models \bigcirc\phi \Leftrightarrow T_1 \models \phi$ | $\bigcirc\phi$ holds iff $\phi$ holds in the suffix starting from the next event of $T$ |
| Eventually | $\Diamond$ | $T \models \Diamond\phi \Leftrightarrow \exists x.(x \geq 0 \wedge T_x \models \phi)$ | $\Diamond\phi$ holds iff $\phi$ holds for some suffix of $T$ |
| Globally | $\Box$ | $T \models \Box\phi \Leftrightarrow \forall x.(x \geq 0 \Rightarrow T_x \models \phi)$ | $\Box\phi$ holds iff $\phi$ holds for all suffixes of $T$ |
| Until | $\mathcal{U}$ | $T \models \phi\ \mathcal{U}\ \psi \Leftrightarrow \exists x.(x \geq 0 \wedge T_x \models \psi \wedge \forall y.(0 \leq y < x \Rightarrow T_y \models \phi))$ | $\phi\ \mathcal{U}\ \psi$ holds iff $\psi$ holds for some suffix of $T$ and until then $\phi$ holds |

For example, the property $\Box(\mathit{Close} \Rightarrow (\neg \mathit{Read}\, \mathcal{U}\, \mathit{Open}))$ state that each time a *Close* event occurs then *Open* should occur later on and *Read* should not occur until *Open* occurs.

The verification of LTL properties can be achieved through the use of existing model checking and runtime verification tools such as SPIN[4] (offline) or $\text{LTL}_3$ Tools[5] (online).

LTL is well-accepted for specifying properties in the verification of concurrent systems. In particular, LTL is used to express safety properties (stating that something bad never happens), and liveness properties (stating that something good keeps happening) [Sistla 1994]. However there also are limitations in LTL, for example, considering distributed traces, we cannot state the property "there is a possibility that *Open* occurs before *Close*", which requires reasoning about sets of traces. This type of statement is usually stated using a branching-time logic (such as CTL) which makes it possible to reason over different timelines.

Timed LTL (TLTL) [Raskin & Schobbens 1999, D'Souza 2003] is an extension of LTL suited to state time-bounded response properties that are common in real-time systems. For example, in TLTL it is possible to state that "*Read* should occur each five minutes".

### 1.6.4 Structured Assertion Language for Temporal Logic (SALT)

The Structured Assertion Language for Temporal Logic (SALT), proposed in [Bauer et al. 2006], is a general purpose language to specify temporal properties. The main motivation behind SALT is to provide an easier way to specify properties using the modalities of LTL and TLTL.

[4] http://spinroot.com
[5] http://ltl3tools.sourcegorge.net

Specifications in SALT consist of three layers: a propositional layer providing atomic, boolean propositions and operators; a temporal layer encapsulating future and past assertions; and a timed layer adding real-time constraints. In the timed layer, it is possible to specify deadlines setting limits for the specification of properties. For example, `always timed[~c] P` states that `P` must be true within the time bounds `c`.

In the temporal layer, SALT allows the specification of scope operators to analyze properties in specific parts of the traces. For example, the statement `assert P between excl a, excl b` defines that `P` should hold only between the events `a` and `b`. However, it is only possible to specify the parts of the trace in terms of initial and final events (statements such as "`P` is evaluated only for the traces of the Webserver" cannot be expressed in SALT).

The language offers support for parametric properties through the creation of functions. For example, `define resp(X,Y) := Y implies X` defines a function that computes the formula $Y \Rightarrow X$ for any given instance of $X$ and $Y$.

Finally, the authors also provide a mechanism for translating specifications in SALT to formulas in LTL and TLTL. This makes it possible to use existing trace analysis tools in order to perform online and offline verification.

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| SALT | centralized | yes | no | yes | both (external tools) |

### 1.6.5 Test Behavior Language (TBL)

Test Behavior Language (TBL) [Chang & Ren 2007] is a language to specify and validate trace-based properties, dedicated mainly to the verification of large telecommunication systems.

In TBL, properties are specified using patterns consisting of a name and a regular expression. Regular expressions represent the structure that should be matched within a trace. For example the following pattern states that an `Input` event with a parameter (a number representing an identifier) should be followed by an `Output` event with the same parameter.

```
pattern P (id:[0-9]+) {'Input $id' ; 'Output $id' }
```

where the connector ";" states that any number of events may appear between the two events.

Expressions can also define time limits that impose the maximum time for which conditions should be evaluated. For example, the expression `*!Input` represents the longest

sequence of `Input` events and it may never terminate since one can wait for more events to occur. One solution is to use the expression `*!Input in (3600,300)` that terminates the matching in an hour (3600 seconds) or whenever any new events do not occur in a 300 seconds period.

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| TBL | centralized | yes | no | yes | offline |

### 1.6.6 LTL$_3$ and TLTL$_3$

[Bauer et al. 2011] define an algorithm for online verification of properties written in LTL and TLTL. Properties are evaluated using three-value logics, called LTL$_3$ and TLTL$_3$ where properties can be evaluated to either 'true', 'false' or 'unknown'. A property is evaluated to 'unknown' whenever it is not possible to know its value because events in the future may change the value of the property. For example, the property $\phi = \Diamond ev_A$ holds for $T =< ev_A >$, but $\phi$ is unknown for $T' =< ev_B >$ because $ev_A$ may still occur in the future.

The authors also propose the use of a four-value logic for property verification. In this approach, properties can be evaluated to 'true', 'false', 'presumably true' or 'presumably false'. The idea is that, if the result of evaluating a given property is unknown, then the values 'presumably true' or 'presumably false' indicate what would be the result if the execution had been finished. For example, consider the simple property $\phi = \Diamond ev_A$ that holds if $ev_A$ occurs eventually and the trace $T =< ev_B >$. The evaluation of $\phi$ for $T$ is 'presumably false' because if the execution finishes then $\phi$ does not hold for $T$; however future events may change the value of $\phi$. This idea in used in [Falcone et al. 2009] for online verification of security properties written in LTL and TLTL.

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| LTL$_3$ | centralized | no | no | no | online |
| TLTL$_3$ | centralized | no | no | yes | online |

### 1.6.7 RuleR and LogScope

RuleR [Barringer et al. 2010b, Barringer et al. 2007] is a rule-based system dedicated to runtime verification, that was used to support testing of spacecraft flight software for the Mars mission of NASA.

In RuleR, a specification is a set of rules, each one of the form `name:cond -> body` which represents a name (`name`), condition (`cond`) and a body (`body`) indicating that, if the

condition is satisfied then the body should be satisfied. For example, the following rules check that only opened files are closed:

| Name | Condition | Body |
|---|---|---|
| `Start:` | `openFile(f:obj)` | `-> Track(f);` |
| `Track(f:obj):` | `!closeFile(f)` | `-> Track(f);` |
| `Close:` | `closeFile(f:obj),!Track(f)` | `-> print('Error in:' + f);` |

The rule `Start` states that once `openFile` occurs the rule `Track` is activated. The rule `Track` states that the rule stays active while `closeFile` does not occur. Finally, the rule `Close` states that if `closeFile` occurs and the rule `Track` is not active then an error is printed.

The authors also show that RuleR can express a wide range of temporal logics (such as LTL) and they provide a prototype implementation of a verification algorithm that can be applied either offline or online. Similarly to $LTL_3$ (Section 1.6.6), to perform online verification RuleR uses a four-value logic, where the values 'still true' and 'still false' correspond to the values 'presumably true' and 'presumably false' respectively. Additionally, a specification is evaluated to 'unknown' if there are some rules evaluated to 'still true' and others to 'still false' at the same time.

LogScope [Barringer et al. 2010a] is an adaptation of RuleR providing a higher-level language to express temporal properties. This language makes it possible to represent conditions expressing that an event should occur eventually (noted `ev`), or an event should not occur (noted `!ev`), or a set of events should occur in a specific order (noted `[ev1,ev2,...]`) or unordered (noted `{ev1,ev2,...}`). For example, the following rule states that if `openFile` occurs then `readFile` and `writeFile` should occur in any order, and these events should be followed by `closeFile`:

```
VerifyRead:  openFile -> [{readFile,flushFile},closeFile]
```

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| RuleR | centralized | yes | no | no | both |
| LogScope | centralized | yes | no | no | offline |

### 1.6.8 ForSpec and Property Specification Language (PSL)

ForSpec [Armoni et al. 2002] is a formal framework, proposed by Intel, for hardware verification. ForSpec includes a language to specify properties based on a combination of LTL and regular expressions. Properties may specify temporal constraints such as event

ordering or deadlines and the language also support parametric properties. Since ForSpec has been designed for hardware verification, it also provides support to specific modalities concerning clock signals. For example, the property `accept_on a P` states that the value of the property `P` should hold until the arrival of the clock signal `a`.

A particular feature of ForSpec is that each property may advance according different clocks and it is possible to specify the sub-traces for which the property should be evaluated. For example, the property `change_on c P` states that the property `P` should be evaluated for the traces defined by the high phases of the clock `c`, i.e. to evaluate `P` it is necessary to use the traces relative to the clock `c`.

Property Specification Language [Vardi 2008] (PSL) is an extension of ForSpec that, among other features, includes a branching-time extension that makes it possible to state properties about multiple timelines. For example, the property `EF P` states that *there is a possibility* that `P` holds in the future. This extension is mainly applied to model checking for the verification of deadlocks properties. However, there is no mention of distributed traces where the total order of the events is unknown.

Initially, ForSpec and PSL were designed with offline verification in mind. In [Morin-Allory et al. 2007] the authors propose an algorithm to build monitoring mechanism for online verification of properties in PSL. Similar to $LTL_3$ (Section 1.6.6), these mechanisms adopt a four-value logic where properties can are evaluated to 'presumably true' or 'presumably false' whenever the evaluation of the property can change due to future events.

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| ForSpec | centralized | yes | yes | yes | offline |
| PSL | centralized | yes | yes | yes | both |

### 1.6.9 The `reduce` Approach

In [Garg et al. 2011] the authors propose an approach to verify traces in compliance with privacy and security policies, which are represented by sets of logical properties. Every event in the trace is associated with a timestamp indicating when the event occurred. For example, the event `send(S,R,'open',5)` means that $S$ has sent to $R$ message '*open*' at time 5. It is assumed that events occurs on distinct times and the total order between events can be obtained using the timestamps.

Properties are expressed using a logic that includes the main modalities of propositional logic. To ensure decidability and efficiency of the verification algorithm, the authors impose that bounded quantifications take the forms $\forall x.(c \Leftrightarrow \phi)$ and $\exists x.(c \wedge \phi)$, where $c$ defines the scope of the variable $x$ and it has a limited syntax where quantifiers and implications are not allowed. In [DeYoung et al. 2010], the authors show that this limited logic make it

possible to express temporal propositions including all modalities of LTL with the addition of real-time constraints.

The main contribution of this approach is a procedure to evaluate properties for incomplete traces, named `reduce`. The verification is based on a three-value logic, similar to $LTL_3$ (Section 1.6.6). However, besides evaluation of properties future events, a property may also evaluate to 'unknown' due to *spatial incompleteness* and *subjective incompleteness*. Spatial incompleteness happens when some traces are stored on non-available physical sites. For example, the property "the Webserver receives the request" may be evaluated to 'unknown' if the trace of the Webserver is not available. Subjective incompleteness happens when some some predicate of the properties rely on human judgment. For example, a property stating conditions about personal medical information of patients may be evaluated to 'unknown' because such information is not present in the traces due to privacy issues.

The `reduce` procedure works by reducing a policy $\phi$ for a trace $T$ to either a boolean value or to a reduced version of $\phi$, which contains only predicates that evaluate to unknown. For example, consider the property $\phi$ = `exists t.(t < 5 and send(S,R,'open',t) and send(S,R,'close',t))` and the trace $T$ = `<send(S,R,'open',4)>`. The result of applying `reduce`$(\phi, T)$ is the reduced form $\phi'$ = `exists t.(t < 5 and send(S,R,'close',t))` since it remains unknown if `send(S,R,'close',t)` may still occur. The resulting policies can be verified incrementally with additional traces until the point that true/false evaluation is obtained or the policy should be analyzed by an expert (in case of subjective incompleteness).

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| `reduce` | centralized | no | no | yes | offline |

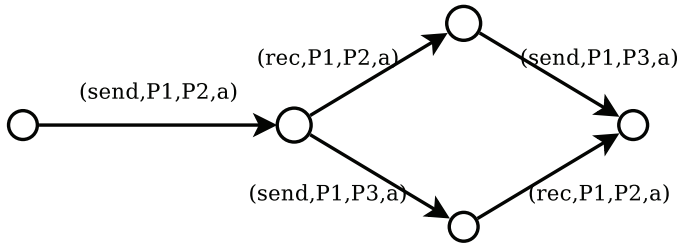
### 1.6.10 ObjectGEODE

[Hallal et al. 2003, Hallal et al. 2006] describe ObjectGEODE, a tool used by Siemens to analyze communication protocols properties. One of the key features of this approach is the verification of properties for distributed traces.

In a trace, events take the form $(send, P_i, P_j, m)$ representing $P_i$ sending message $m$ to $P_j$, $(rec, P_i, P_j, m)$ representing $m$'s reception, or $(rdv, P_i, P_j)$ for a synchronization (rendezvous) event between $P_i$ and $P_j$. Traces are represented by automata where state transitions correspond communications between processes and are labeled by the corresponding event. For example, consider the following traces:

$$\begin{array}{ll} Trace_1\text{:} & (send, P_1, P_2, a), (send, P_1, P_3, a) \\ Trace_2\text{:} & (rec, P_1, P_2, a) \end{array}$$

These traces are represented by the following automaton:




This automaton-based specification allows the use of model checking tools to produce the *scenarios* that correspond to a possible total order of the events. For example, the above automaton produces the following two scenarios:

$$\begin{array}{ll} Scenario_1\text{:} & (send, P_1, P_2, a), (send, P_1, P_3, a), (rec, P_1, P_2, a) \\ Scenario_2\text{:} & (send, P_1, P_2, a), (rec, P_1, P_2, a), (send, P_1, P_3, a) \end{array}$$

The verification of a property for a set of traces consists of computing all scenarios and evaluating the property for each scenario. The result of the evaluation is two sets containing the scenarios where the property holds or not.

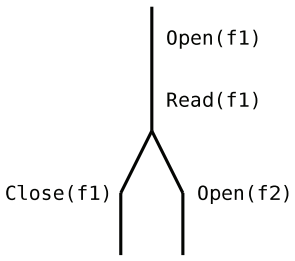
The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| ObjectGEODE | distributed | no | no | no | offline |

### 1.6.11 Tree-based Analysis of Traces

[Saleh et al. 2007, Arasteh et al. 2007] define a framework for forensic analysis based on a branching-time logic to verify properties for distributed traces.

Traces are represented by trees where branches represent concurrent sequences of events. For example, the following tree defines a distributed trace where the events `Close(f1)` and `Open(f2)` occur concurrently:




Properties can include modalities of branching-time logic that makes it possible to reason over the various scenarios that may be produced from the tree. For example, it is possible to state that "in at least one of the scenarios `Close` occurs before `Open`".

Temporal conditions are expressed using patterns of events and parameters to be matched in traces. For example, the property `<x.Open(sID).x.Close(sID).x>` checks if an event `Open` occurs followed by an event `Close` both with the same parameter. The symbol `x` works as a wildcard to indicate that any number of events may occur before, after or between the two events.

One advantage of this logic is the possibility to state properties that should be verified for specific portions of the traces. For example, the property `«Server»<x.Close().x>` checks if `Close` occurs in the traces of the server.

The features of this proposal are summarized in the following table:

| Proposal | Trace structure | Parametric properties | Sub-trace verification | Timed Verification | Online/Offline Verification |
|---|---|---|---|---|---|
| Tree-based Analysis | distributed | yes | yes | no | offline |

### 1.6.12 Summary

We conclude that the analysis of distributed traces consists mainly of computing the scenarios representing the possible total orders of the events, where the same property is evaluated for every scenario, such is the case of ObjectGEODE (Section 1.6.10). A more complete solution consists of providing a branching-time logic (such as CTL) where is possible to reason about the different scenarios computed, such is the case of a tree-base analysis (Section 1.6.11).

One of the key issues in the analysis of distributed traces is to be able to deal with the potentially exponential number of scenarios that may be computed. *Computational slicing* [Sen & Garg 2003, Mittal & Garg 2001] has been proposed to address this issue. This approach consists of building selected parts of the traces which contain only events that may change the evaluation of the property, which may considerably reduce the number of scenarios.

As far as efficiency is concerned, the support of parametric properties can also make the analysis much more expensive. In some cases, such as earlier versions of LogScope (Section 1.6.7), the support to these feature may be limited in return for better performances of the verification procedure.

In online evaluation, the evaluation can remain inconclusive because it depends on the occurrence of future events. One solution is to use a three-value (or four-value) logic for which the value 'unknown' indicates that a property may still change with the advent of future events. Another solution is to impose a time limit stating when the property should be verified, such as in TBL (Section 1.6.5). The `reduce` approach (Section 1.6.9) extends the verification of properties with future incompleteness to also evaluate properties to 'unknown' when some events are not available due to other reasons, such as location access or privacy issues.

Finally, among the proposals mentioned here, only PSL (Section 1.6.8) and tree-based analysis provide support for the analysis of sub-traces. However, these solutions do not offer mechanisms verify that only the sub-traces specified are used in the evaluation of the property.

## 1.7 LISE Project

LISE[6] (Liabilities Issues in Software Engineering) is a multidisciplinary project funded by the French National Research Agency (ANR-07-SESU-007). The project is led by INRIA and involves two research groups in law and four research groups in ICT.

This project addresses the problem of liabilities in the restricted context of B2B contracts concerning computer system products. In contrast with forensics, where the approach is to look for evidence after a problem has occurred, a contractual framework allows the parties to consider an *a priori approach* where dysfunctions, liabilities and electronic evidence are defined before the delivery of the system.

The objective of the project is to elaborate a methodology for assisting parties to elaborate such B2B contracts and to ensure that convincing digital evidence will be available to establish liabilities in case of failure. The approach will also facilitate an amicable and precise settlement of liabilities between the parties, avoiding for instance excessive and stronger liabilities exemptions that could be challenged or invalidated in court [Genicon 2008, Bitan 2004]. In fact, the necessity to resort to a judge comes only from the lack of agreement between the parties about the consequences of system failures and potential compensations. If the contract is precise, coherent and balanced enough w.r.t the share of liabilities, the legal costs can be considerably reduced.

The LISE methodology includes *technical and legal solutions* for stating contractual liabilities in an integrated way. By technical solutions we mean a set of tools supporting the parties for describing and evaluating liabilities in a precise and unambiguous manner. By legal solutions we mean a contractual framework which is in conformance with the applicable law and jurisprudence. Legal solutions take the form of a set of contractual clauses referring to technical annexes, which specify liabilities, electronic evidence and application of liabilities.

In the following chapter, starting from the legal solution elaborated by the lawyers of the project, we exhibit the expected requirements of the technical framework, which is developed in the sequel.

[6] http://licit.inrialpes.fr/lise/

# Chapter 2

# LISE Approach

As described in the previous chapter, we focus on *business-to-business* (B2B) contracts dedicated to the development or integration of computer systems. We mainly focus on issues related to failures of computer components. In particular, we do not take into account liabilities for delays in service delivery or intellectual property right's infringements.

As an illustration, consider a car embedded system controlling an automatic urgency braking mechanism based on the recognition of obstacles[1]. This system may be the result of the combination of various computer components (such as the obstacle detection software, the braking activation mechanism, the alert system, etc.) that are supplied by contractors and put together by an integrator. The scope of the contract should describe the liabilities of the parties in case of a failure of the system.

We focus on B2B contracts because legal constraints for these types of contracts are not as strong as for *business-to-consumer* (B2C) contracts, generally subject to specific legal protections of consumers. In B2B contracts, the parties are considered is principle as equal in power and able to understand the implications and the scope of their commitments. In contrast, in B2C contracts, some clauses can be considered unfair considering the lack of knowledge and expertise of average customers.

In this chapter, we describe in more detail our approach to help in the elaboration of the parts of B2B contracts which are dedicated to system failures and the resulting liabilities. First, we describe the objectives of our approach and introduce our terminology (Section 2.1) before sketching the legal framework proposed by the lawyers of the LISE project to achieve the objectives (Section 2.2). Then, we proceed with the requirements and functionalities of a technical framework to be used to elaborate the parts of the contract dedicated to liabilities for computer system failures (Section 2.3). Finally, we review the technology used to implement our framework (Section 2.4).

[1]For instance, tests performed on the braking system of the Volvo S60 show that the risks of failures are real. More information on http://carscooop.blogspot.com/2010/05/epic-fail-2011-volvo-s60-warning-with.html

## 2.1 Objective of Our Approach

The objective of our approach is to provide a well-understood and non ambiguous procedure allowing the parties to establish liabilities for damages caused by failures of the system. To this aim, we provide a procedure based on the analysis of the logs of the system. By procedure, we mean not only the tools used in the log analysis but also the way to use these tools and the actors in charge of each step of the analysis.

In order to propose a reliable and trusted procedure to define and establish liabilities, two main requirements must be satisfied. First, there should exist a conclusive and reliable way to identify the component that caused the failure. Second, there should exist a precise method, accepted by the parties, that (based on the observation of the erroneous components) leaves no doubt about the liabilities.

Considering our objective and these requirements, the contract must include the following elements:

1. A definition (as precise as possible) of the system and its components.
2. The identification of failures considered important enough to warrant a formal definition of the associated liabilities.
3. The definition of logs that can be used as digital evidence.
4. The relationship between the components of the system and the liable parties.
5. The definition of the procedure to establish liabilities and the actions to follow when the proposed approach is not applicable, such as a failures not initially foreseen.

The aim of the LISE approach is to propose an integrated framework to help the parties to elaborate the contract containing the elements mentioned above. In Section 2.2, we introduce the model of the contractual provisions dedicated to liabilities proposed by the lawyers of the LISE project [Steer et al. 2011]. This model consists of a set of legal clauses which refer to technical annexes. The aim of this chapter is to define the content of these technical annexes as precisely as possible, in particular pointing out elements that have to be formalized and proposing a well-defined procedure for the log analysis. Before entering into this description, we introduce some terminology.

### 2.1.1 Terminology of Computer System Misbehavior

We adopt a standard terminology [Avizienis et al. 2004] to define computer system misbehaviors:

- A *system* is "an entity that interacts with other entities, i.e., other systems, including hardware, software, humans and the physical world. [...] A system is composed of a set of *components* bound together to interact".

- A *system failure* (or simply *failure*) is "an event that occurs when the delivered service deviates from correct service". That is, a failure is a transition from a correct behavior to an incorrect behavior.

- An *error* is "the part of the total state of the system that may lead to its subsequent service failure. [...] many errors do not reach the system's external state and cause a failure". That is, a failure occurs when an error reaches the service interface of the system.

- A *fault* is "the adjudged or hypothesized cause of an error. [...] A fault is active when it causes an error, otherwise it is dormant."

For example, a fault may be a missing variable initialisation. This fault becomes active and causes an error when one of the components tries to access the value of the variable. A failure occurs for instance if an access to an uninitialized variable raises an exception, which makes the system stop and display an error message to the user.

### 2.1.2 Terminology About Logs

We adopt the following terminology based on the NIST Guide to Computer Security Log Management [Kent & Souppaya 2006]:

- A *log file* (or simply a *log*) is a record of the *events* occurring within a system. Logs are composed of *log entries*, each one containing information concerning a specific event that occurred in the system.

- A *log infrastructure* consists of the material (hardware and software) used to generate, transmit, store and dispose the log data.

- A *log management policy* is the process established in an organization for generating, transmitting, storing and disposing log data.

## 2.2 LISE Contract

The model of contract proposed in [Steer et al. 2011] is divided into articles, each one composed by a set of legal clauses defining the terms of the contract. The first article

provides the definitions used in the rest of the contract[2]:

**Article 1: Definitions**

**Computer System**: The Computer System subject to this Agreement is the integration of the set of computer components as specified in Annex A.

**Contract**: This Agreement includes the present document and its annexes A, B, C, D and E.

**Log**: The Logs are the files recording the events occurring within the Computer System.

**Log Analysis Procedure**: The Log Analysis Procedure is the procedure defined in Annex C and applied by the Parties to collect and analyze the information contained in the Logs in order to establish liabilities as set forth in Article 3 of the Agreement.

**Log Analyzer**: The Log Analyzer is the software tool defined in Annex D and used in the Log Analysis Procedure in order to establish the validity of a claim and to identify the components involved in the handling of the claim.

**Log Infrastructure**: The Log Infrastructure is the set of tools used to generate, transmit, store and dispose the Logs as specified in Annex B.

**Party**: A Party is any entity signing this Agreement.

**Provider**: The Provider is the Party in charge of providing the Computer System to the customer.

[...]

The model of contract includes three other articles. Article 2 describes the agreement between the parties to recognize the legal value of the logs as digital evidence. Article 3 defines the liabilities associated to each party. Finally, Article 4 defines the actions that should take place in order to establish the liabilities in case of failure. These articles refer to five annexes providing the technical details of the contract:

- Annex A provides a description of the system and failures for which liabilities are defined.
- Annex B describes the log infrastructure and a log management policy.
- Annex C defines the log analysis procedure.
- Annex D defines the tool used in the log analysis procedure (log analyzer).
- Annex E defines the liability relationship (associating combinations of failures and liable parties).

In the following sections, we discuss in more detail each part of the contract in association with the content of these annexes. We use the car braking system case study to illustrate our approach.

[2] Translated to English from the original in French [Steer et al. 2011]

### 2.2.1 Specification of the System (Annex A)

Annex A is mentioned in Article 1 and it provides a description of the system in terms of its expected functionalities, its architecture, the behavior of each of its components and its potential failures. Annex A also specifies the components that should be logged and for each of these components, the information that should be logged. Annex A should thus include:

1. The specification of the system.

2. The identification of failures for which the parties wish to specify liabilities.

3. The specification of the information that should be recorded in the logs.

The specification of the system and failures can be used to check whether or not the system behaved as expected. The failures are expressed in terms of information recorded in the logs. For example, we can specify that the braking system fails either because the obstacle detection component did not communicate with the braking component or the braking component did not activate the brake in time. To specify this failure, we define the actions executed by the two components and the expected exchange of information between these components that are recorded in the logs. It is also necessary to specify which components should be logged and the precise information that should be logged.

Ideally, to ensure an unambiguous and precise description of the system and the liabilities it is recommended to use rigorous techniques for the description of the computer system (such as semi-formal or formal methods). In our framework we provide a way to specify the information that should be recorded in the logs and, based on this information, a formal way to describe the failures.

### 2.2.2 Definition of Evidence (Article 2)

Article 2 defines the agreement about the logs that have to be supplied by the parties and how they will be used to establish liabilities. More precisely, the clauses of this article specify the agreement about:

- The log infrastructure and log management policy (clauses 2.2 and 2.3).

- The procedure and tools that should be used when a failure is reported (clause 2.4).

- The admissibility and probative value of the results of the log analyzer (clause 2.5).

The article is structured as follows:

**Article 2 : Definition of Evidence**

**2.1** The Parties hereby accept and agree to apply the rules defined in this article to define the evidence to be used to establish their liabilities as set forth in Article 3.

**2.2** The Logs are recorded by the Log Infrastructure in the format specified in Annex B and stored on the devices of the Parties (or third parties) set forth in Annex B during (x) months. After this (x) months period, to the extent not prohibited by law, the Logs shall be deleted by the aforementioned Parties (or third parties).

**2.3** The Parties set forth in Annex B shall use the Log Infrastructure to record the Logs as specified in Annex B. They commit not to modify, delete or alter log entries in any way. When Annex B provides for a third party to host the Logs, the Parties hereby agree to accept the aforementioned third party as an independent escrow and commit to send him the execution data necessary to build the Logs. The execution data sent by the Parties shall be complete (as required by Annex B) and unmodified.

**2.4** The Parties hereby agree to apply the Log Analysis Procedure and the Log Analyzer in the following events:

- Failure of the Computer System observed or suspected by a Party.
- Claim of a customer against one of the Parties about an alleged failure of the Computer System.

**2.5** The Parties hereby agree to grant to the results of the Log Analysis Procedure the weight of evidence and legal value. They commit not to challenge the acceptability, accuracy or probative value of the results of the Log Analyzer except in case of act in bad faith or intentional fault of one of the Parties.
[...]

As pointed out in clauses 2.2 and 2.3, Annex B should include:

1. The log infrastructure.
2. The log management policy (including the parties involved).
3. The log distribution (including a description of the components responsible for logging).

As mentioned in Section 1.5, to ensure the probative value of log based evidence, the log infrastructure and a log management policy should be defined precisely. Certain aspects of the log infrastructure may be specified by a logging protocol, such as the transmission protocol for log data. Due to the potentiality distributed nature of the system, an important information to be specified is the *log distribution* describing how entries are distributed

among the various logs files and the logging components. For example, in a client-server architecture the log distribution may be composed of a simple log file containing entries of all the clients (logged by the webserver which is the contact point with the clients) and another log file to record the information of the server (logged by the central server).

### 2.2.3 Definition of Liabilities (Article 3)

Article 3 defines the agreement about liabilities in case of failure and the compensations associated with these liabilities (clause 3.1). This agreement excludes liabilities related to criminal actions or defective products (clause 3.2) because the law provides specific rules for such cases[3]. The article is structured as follows:

**Article 3 : Definition of Liabilities**

**3.1** The Parties commit to apply the rules defined in Annex E for the definition of liabilities following the procedure defined in Article 4 and shall not object to their conclusions. The identification of the component(s) involved in the failure of the Computer System by the binding procedure defined in Article 4 shall be used to identify the liable Parties as defined in Annex E and the aforementioned liable Parties shall indemnify the plaintiff in the conditions and within the limits set forth in Annex E.

**3.2** The liability rules defined in Annex E shall apply only to the extent not prohibited by law; they shall not apply to tort liability or bodily injuries.
[...]

Annex E may take the form of tables that define the parties liable for a given failure and the associated compensations. The following table defines the assignment of liabilities for a failure in the braking system:

| **Annex E: Assignment of liabilities** | | | | | |
|---|---|---|---|---|---|
| **If errors have occurred in:** | **Then, the liabilities will be assigned to:** | | | | |
| | P1 (supplier of obstacle detection software) | P2 (supplier of brake component) | P3 (supplier of obstacle detection hardware) | P4 (system integrator) | P5 (system supplier) |
| Obstacle detection software | X | | | | |
| Obstacle detection hardware | | | X | | |
| Brake component | | X | | | |
| Obstacle detection functionality | X | | X | | |
| Obstacle detection functionality due to extreme weather conditions | | | | | X |
| Integration between components (bad interface definition) | | | | X | X |
| No identified component | | | | | X |

[3]For instance, Article 1150 of the French Civil Code states that clauses limiting liabilities will be considered null if they concern corporal damages or defective products.

The first column contains the components for which an erroneous behavior is observed in the logs and the other columns define the parties which will be held liable for the failure. Some lines (such as the first line) simply allocate the liability to the provider of the faulty component. Other lines (such as the fourth line) define several liable parties because the information available is not precise enough to identify a single party. Whatever the underlying motivations, the point of that these allocation of liabilities must come from the agreement of the parties. The table specifying the compensations is specified as follows:

| **Annex E: Compensations** | |
|---|---|
| **If errors have occurred in:** | **then the liability shall amount to:** |
| Obstacle detection software | Full amount of the damages caused by the failure |
| Obstacle detection hardware | Full amount of the damages caused by the failure |
| Brake component | Full amount of the damages caused by the failure |
| Obstacle detection functionality | Half of the amount of the damages caused by the failure for each of the designated Parties |
| Obstacle detection functionality due to extreme weather conditions | No damages |
| Integration between components (bad interface definition) | Indemnification by P4 up to (x) EUR, and by P5 for any damage in excess of (x) EUR. |
| No identified component | Full amount of the damages caused by the failure |

In Chapter 4 we provide a formal way to specify certain aspects of the liability relation defined in the above tables.

### 2.2.4 Definition of the Claim Handling Procedure (Article 4)

Article 4 defines the actions that should be taken to establish liabilities when a failure has occurred or a claim is raised by a third party with respect to the system. The first clauses (4.1, 4.2 and 4.3) define how the parties should notify a failure and the corrective actions:

**Article 4 : Claim Handling Procedure**

**4.1 Notification of the claims.**
The Provider commits to apply the procedure set forth in this Article 4 as soon as he receives a claim notification from a customer for an alleged failure of the Computer System. If the claim is notified by the customer to another Party, the Party receiving the claim shall apply the same rules and comply with this article as if it were the Provider for the purpose of this Article 4.

**4.2 Information.**
The Provider shall notify the claim and all relevant information pertaining to the claim to the other Parties without delay (and the latest one day after receipt of the notification of the claim). Notification shall be sent by registered post with acknowledgment of receipt.

**4.3 Corrective actions.**
After receipt of a claim notification, the Parties (or, if applicable, the Parties specified in the Log Analysis Procedure) commit to take without delay all appropriate corrective actions in order to repair the Computer System and limit the damages.
[...]

The notification of the failure starts with a *claim*. In our context, a claim is a complaint, either legal or amicable, from a party (called the *plaintiff*) against another party (called the *defendant*). Claims can also be initiated by third parties, such as the consumer in a B2B contract.

The following two clauses (4.4 and 4.5) define the agreement concerning the process of collecting and analyzing the logs to handle a claim:

**4.4 Collection of the Logs.**
After receipt of a claim notification, the Party specified in Annex C shall without delay (and the latest within 1 day after receipt of the notification) apply the procedure set forth in Annex C to collect and protect the Log data. If applicable, the aforementioned Party shall also forward the Log data to the trusted third party specified in Annex C in the conditions defined in Annex C.

**4.5 Log Analysis**
The Parties shall apply the Log Analysis Procedure set forth in Annex C without delay (and the latest within 3 days after receipt of the claim notification) and use the results of this procedure to decide upon the validity of the claim and, if applicable, their respective liabilities for the claim following the rules defined in Annex E. The Provider shall notify the result of the application of the Log Analysis Procedure to all the Parties without delay (and the latest within two days after the availability of these results).
[...]

As pointed out in the above clauses, Annex C should include:

- The log collection procedure.
- The log analysis procedure.

More precisely, Annex C specifies the party (or an expert) that should be responsible for conducting the claim handling procedure. The claim handling procedure starts with the collection of the logs. This step should comply with precise requirements (specified in Annex C) to guarantee the integrity of the logs. For example, it may be required that the logs to be analyzed should be transmitted using a secure protocol to preserve their privacy and integrity. Another part of Annex C specifies the *log analysis procedure*. The objective of this procedure is to identify the failing components relying on the logs collected and the definitions of failures in Annex A.

Annex D specifies the *log analyzer* be used in the log analysis procedure. Given the specification of the failures in Annex A and the logs, the log analyzer checks if the failure indeed occurred and the potential errors that may have caused the failure.

Finally, the remaining clauses of Article 4 define what should be done when the results of the log analysis are challenged or are not sufficient to establish liabilities. In this case the parties should resort to a third party, typically an expert, to solve the issue. This expert may use the logs (and any other resources) to find the component that caused the failure and to designate the liable parties.

## 2.3 Technical Framework

Our objective is to define a method to specify contractual liabilities based on the model of contract presented in the previous section. To this aim, we propose a framework consisting of a set of tools that can be used to elaborate certain aspects of the technical annexes of the contract.

One requirement of our framework is to allow to precisely specify the liabilities for a given set of failures and the process to establish these liabilities using the logs. To reach this goal, we need to specify the logs, including their content and distribution, as well as the log analysis tool.

Another requirement of our framework is to provide a support for contract simulation, i.e. the ability to validate the results of the liabilities for given logs. This includes the verification of properties of the logs, the analysis of log distribution and the animation of claim scenarios.

In the rest of this thesis we present a formal framework to specify liabilities that fulfills these requirements. More precisely, our framework is based on a set of formal model templates that can be completed on a case-by-case basis. These models specify not only the logs but also the failures for which the parties wish to establish liabilities.

Our framework includes the following:

- A model to specify the API of the components (Annex A).
- A model to specify the logs: the content of the entries (Annex A) and their distribution (Annex B).
- A model to specify the failures of the system considered in the contract (Annex A).
- A model to specify the assignment of liabilities (Annex E)
- A specification of the log analyzer (Annex D) and the claim handling procedure (Annex C).

The benefits of a formal language include a well-defined semantics that allows us to precisely specify the log content and its analysis to establish liabilities. Besides this, it is also provides a support to formally verify implementations of this analyzer. Yet another advantage is the possibility to validate the contract using tools that support verification and animation of models.

Our methodology (Figure 2.1) is based on a set of formal model templates to be instantiated by the parties (or experts representing the parties). The aim is to provide a structured and precise way to specify the information of the annexes. The models also contain certain properties about the logs and the system that the parties wish to verify precisely. For example, the parties may state properties about the logs to verify their integrity.

Using model verification tools it is possible to verify if the properties in the model hold for a specific case and a given set of logs. Tools can also be provided to support the animation of possible scenarios. This process is usually performed by an investigator that can be either the parties or a third party (expert) designated to analyze the logs.

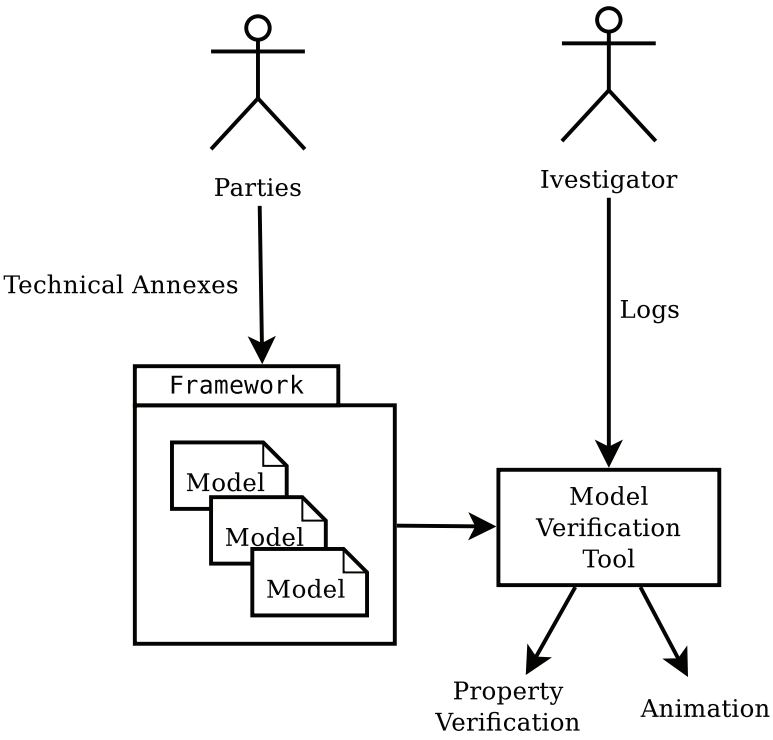



Figure 2.1: Framework Methodology

We have chosen to implement our framework using the B-method. In the following section we describe the benefits of this approach and review the main concepts of the

technology.

## 2.4 Background on the B-Method

The B-method [Abrial 1996] is a formal method of software development that has been successfully applied in safety-critical systems in academy and industry [Behm et al. 1999, Badeau & Amelot 2005, Rehm 2010]. The B method encompasses activities of modeling, safety properties verification and refinement until code generation and is supported by an industrial tool [ClearSy 2011] and an open platform [Rodin 2011]. These tools assist the development process in its whole, from specification until code generation. Some tools are dedicated to validation of models, such as animators (ProB [Leuschel & Butler 2003], Brama [Servat 2007]) and test generators (jSynoPSys [Dadeau & Tissot 2009], Leirios [Jaffuel & Legeard 2007]).

In the LISE context, we have chosen the B-method for several reasons:

- the industrial status of this approach is a positive argument in terms of trust, a important notion in our context.
- the B-method is based on a first order set theory which is understandable and manipulable way by computer science engineers who can be implied in the technical content of contracts.
- the maturity of tools allows us to use them during contract elaboration in order to validate liability specification on concrete cases.

In this thesis we exploit a very restricted part of the B method: we are mainly interested to model data and to state some verifications and calculus on them. In particular the refinement facility will be not used.

### 2.4.1 Abstract Machines

In B-method, each model is represented by an *abstract machine* that defines the state of the system, properties about this state and how it evolves. Abstract machines take the

following form:

```
MACHINE M
SETS S              /* given and enumerated sets */
CONSTANTS C         /* list of constants */
PROPERTIES P        /* properties in the form of a first order formula */
VARIABLES V         /* list of variables */
INVARIANT I         /* a first order formula stating properties
                       on variables and constants */
INITIALISATION U    /* initial assignment of variables */
OPERATIONS          /* operation definitions in the form of
...                    pre and postconditions /
END
```

Properties and invariant are stated using a first order set typed theory (see section 2.4.4). Clause SETS allows to introduce enumerated and given sets, that are considered as type. Initialisations and operations are stated using the generalized substitution language, a specification language based on assignment. However, we will not describe this language here because it will not be used in the context of this thesis.

**Example 2.1.** Example of abstract machine

Machine *BreakController* defines the actions (enumerated set $ACTION$) and parameters (given set $PARAM$) that define the API of a break controller component. The constant $NumParams$ maps each action into the number of parameters of the action, that should be greater that one ($c1$).

MACHINE *BreakController*
SETS $ACTION = \{Activate, Deactivate, EmmitAlert\}; PARAM$
CONSTANTS $NumParams$
PROPERTIES
$NumParams \in ACTION \rightarrow \mathcal{N} \wedge$
$NumParams = \{(Activate \mapsto 3), (Deactivate \mapsto 2), (EmmitAlert \mapsto 2)\} \wedge$
($c1$) $\forall action.(action \in \mathsf{dom}(NumParams) \Rightarrow NumParams(action) > 1)$
END

The operator $\mathsf{dom}(f)$ returns the domain of the function $f$ (see Section 2.4.4).

### 2.4.2 Machine consistency

Machine consistency is established using proof obligations. Verification imposed in [Abrial 1996] is relative to invariant preservation: we have to prove that the invariant is

established by initialisation and preserves by each operation. Due to the fact that our models do not contain variables we not detailed these proof obligations here. Another possible verification consists to establish property consistency, as proposed in [Schneider 2001]:

$$H_S \Rightarrow \exists \mathbf{C}.\mathbf{P}$$

where $H_S$ are the hypothesis relative to given and enumerated sets. For example if a machine contains the given set $S$ and the enumerated set $T = \{a, b\}$ than $H_S$ is equivalent to:

$$S \in \mathcal{P}_1(INT) \wedge T \in \mathcal{P}_1(INT) \wedge T = \{a, b\} \wedge a \neq b$$

The B-method does not impose this verification at the abstract machine level. In fact, feasibility properties are normally automatically established at the level of implementation: if we build a correct implementation the existence of constants and variables is proved. In particular, an implementation must contain a clause called VALUE that explicitly assigns values to constants. This clause generates a proof obligation that verifies all properties starting from the abstract machine level until the implementation. However, we to not use the refinement process in this thesis.

### 2.4.3 Structuring Machines

Models can be structured to constitute larger models using INCLUDES and SEES clauses.

#### The INCLUDES clause

If $M_2$ *includes* $M$ (the machine in Section 2.4.1) then it means that $M_2$ contains a copy of $M$, i.e. sets, constants and variables of $M$ are included into $M_2$. We specify $M_2$ as follows:

| |
|---|
| MACHINE $M_2$ INCLUDES $M$ |
| SETS $S_2$ |
| CONSTANTS $C_2$ |
| PROPERTIES $P_2$ |
| VARIABLES $V_2$ |
| INVARIANTS $I_2$ |
| INITIALISATION $U_2$ |
| OPERATIONS ... |
| END |

Variables defined in $M$ can only be assigned using a call of M's operations (encapsulated use of $M$). Thanks to this restriction, the invariant of $M$ is preserved by construction. Proof

obligations attached to $M_2$ are only dedicated to the fact that $I_2$ is respected by the initialisation $U_2$ and new $M_2$ operations. Satisfiability proofs, as proposed in [Schneider 2001], is the following one:

$$H_S \wedge H_{S_2} \Rightarrow \exists C, C_2.(P \wedge P_2)$$

Therefore, if $M_2$ is consistent (Section 2.4.2) then $M$ is also consistent.

#### Structuring with SEES

If $M_2$ *sees* $M$ then this represents that $M_2$ can read the data of $M$ but $M_2$ cannot modify the data of $M$. We specify $M_2$ as follows:

| MACHINE $M_2$ SEES $M$ |
|---|
| ... |
| END |

Constants of $M$ can be used to define the types and values of constants in $M_2$. The difference between INCLUDES and SEES is that when structuring with SEES the variables of $M$ are not referenceable within the invariants of $M_2$ and the operations of $M$ are not referenceable within the operations of $M_2$ (only read operations). However, since the machines presented in this thesis do not contain variables we only structure machines using SEES. Unlike the *include* relationship, the *see* relationship is not transitive, therefore if a machine $M_3$ sees $M_2$ then $M_3$ cannot read the sets and constants of $M$.

### 2.4.4 B set theory

The basic set constructs are the following:

| Construct | Name |
|---|---|
| $S \times T$ | Cartesian product |
| $\mathcal{P}(S)$ | Powerset |
| $\{x \mid P\}$ | Set comprehension |
| $BIG$ | An infinite set |

The cartesian product $S \times T$ of two sets are the set of ordered pairs $(s, t)$ such that $s \in S$ and $t \in T$. The powerset $\mathcal{P}(S)$ of a set $S$ is the set of all subsets of $S$. In B is also possible to use $\mathcal{F}(S)$ to represent the set of all finite subsets. Set comprehension $\{x \mid P\}$ is the set of elements for which the condition $P$ is true. Finally, the infinite set $BIG$ allows us to define sets such as integers.

From the construct above we can define new domains and operators such as relations, functions and sequences.

### Relations

Given two set $S$ and $T$ we express a relation $rel$ between these two sets (noted $rel \in S \leftrightarrow T$) as a set of elements of the form $s \mapsto t$ that belongs to the cartesian product $S \times T$:

$$rel \in S \leftrightarrow T \Leftrightarrow rel \subseteq \{(s \mapsto t) \mid s \in S \wedge t \in T\}$$

The domain and range of $rel$ is defined respectively as follows:

$$\begin{array}{c} \mathsf{dom}(rel) = \{s \mid s \in S \wedge \exists t.(t \in T \wedge s \mapsto t \in rel)\} \\ \mathsf{ran}(rel) = \{t \mid t \in T \wedge \exists s.(s \in S \wedge s \mapsto t \in rel)\} \end{array}$$

The *relational inverse* of a relation $rel$ (noted $rel^{-1}$) is defined as:

$$rel^{-1} = \{t \mapsto s | s \mapsto t \in rel\}$$

In a relation $rel$ the operator $rel[U]$ identifies all elements in $\mathsf{ran}(rel)$ that are related with an element belonging to $U$, i.e.:

$$rel[U] = \{t \mid s \mapsto t \in rel \wedge s \in U\}$$

### Functions

A *function* between two sets $S$ and $T$ is a relation which relates the elements of $S$ to *no more that one* element of $T$. Functions can be partial ($fun \in S \nrightarrow T$) or total ($fun \in S \rightarrow T$) depending on their domain, i.e.:

$$\begin{array}{c} S \nrightarrow T = \{fun \mid fun \in S \leftrightarrow T \wedge \forall s, t_1, t_2.(s \in S \wedge t_1 \in T \wedge t_2 \in T \Rightarrow \\ ((s \mapsto t_1 \in fun \wedge s \mapsto t_2 \in fun) \Rightarrow t_1 = t_2)\} \\ S \rightarrow T = \{fun \mid fun \in S \nrightarrow T \wedge \mathsf{dom}(fun) = S\} \end{array}$$

The notation $fun(s)$, with $s \in \mathsf{dom}(fun)$, denotes the value of $fun$ associated to $s$.

We can also state functions using the *lambda notation*. The form of a lambda definition is $fun = \lambda s.(s \in S \mid E)$, which maps $s$, of type $S$, to the value of expression $E$. For example, the mathematical function *square* given by the definition $square(x) = x^2$ can be expressed as follows:

$$square = \lambda x.(x \in \mathbb{Z} \mid x^2)$$

The more general form of a lambda notation is $\lambda z.(P \mid E)$ defined for a list of variables $z$, provided that $P$ defines the type of each variable in $z$.

### Sequences

Finally, *sequences* are finite ordered list of elements of a given type that can be understood as a special kind of function that maps natural number into elements of a given set. We denote $\mathsf{seq}(S)$ the seq of possible sequences of the elements of $S$:

$$\mathsf{seq}(S) = \bigcup_{N=0}^{n} (1..N \rightarrow S)$$

Additionally, we use the notation $\mathsf{iseq}(S)$ to denote *injective sequences*, i.e. all sequences of elements of $S$ that contain distinct members. The value of a sequence can be defined by listing its members within square brackets and the number of elements is obtained using the function $\mathsf{size}$. The following example defines a sequence $ss$ of natural numbers that contain three elements:

$$ss \in \mathsf{seq}(\mathcal{N}) \wedge ss = [1, 3, 6]$$

The symbol $\mathsf{union}$ represents the generalized union of all elements of a given set, i.e.:

$$\mathsf{union}(TT) = T_1 \cup T_2 \cup \cdots \cup T_n \text{ such that } T_i \in TT \text{ for } 1 \leq i \leq n$$

For example, let $TT = \{\{1, 2\}, \{3, 4\}, \{5\}\}$ then:

$$\mathsf{union}(TT) = \{1, 2\} \cup \{3, 4\} \cup \{5\} = \{1, 2, 3, 4, 5\}$$

# Chapter 3

# Log Analysis

As mentioned in the previous chapter, we propose a framework consisting of a collection of models to be completed, to assist parties in the elaborating of the content of the technical annexes. In this chapter we present the models used to specify the logs and their analysis.

First, we describe some basic assumptions about the system and the communications between its components (Section 3.1) and we describe the case study, used to illustrate our approach (Section 3.2). Then, we introduce the models used to represent the logs (Section 3.3) and the operations on distributed logs (Section 3.4). Then, we introduce the models allowing us to state properties on the logs and provide a formal specification of the log analyzer to verify properties of distributed logs (Section 3.5). We also define an incremental version of the log analyzer (Section 3.6). Finally, we point out the parts of the technical annexes corresponding to each model (Section 3.7) and we summarize the main contributions of this chapter (Section 3.8).

## 3.1 Assumption of the System and Communications

First, we consider distributed systems consists of components that communicate by exchange of asynchronous messages. These messages may represent exchange of information (either between components or between components and users of the system), method calls or internal actions executed by the components. This approach is a standard way to model interactions of components and to abstract different kinds of communications that may occur in the system [Coulouris et al. 2011]. We assume that the messages are recorded in the chronological order of the communications. This hypothesis is common in trace analysis (Section 1.6).

Second, following the approach taken in the contract formalisms presented in Section 1.4, we assume that the components communicate without loss of information. This assumption is likely to be valid in a B2B contractual context because, in order to establish liabilities, the parties need to distinguish between errors in the com-

ponents and communication errors. More specifically, when an error occurred because a message has not been received by a component it should be possible to establish if the error was caused by the component or by the network. Several logging protocols have been proposed to ensure that communications occur without loss of message [BalaBit IT Security 2011, Kelsey et al. 2009, Sackmann et al. 2006, Ma & Tsudik 2009] (Section 1.5). They rely on communication protocols (such as TCP) that ensure message delivery by retransmission of lost messages.

Third, we also assume that messages exchanged between components are unique and can be distinguished from each other. In particular, in order to recompose logs, we have to detect the pairs of log entries corresponding to the send and receive operations for each message. The same assumption is used in the analysis of distributed traces, such as Object-GEODE [Hallal et al. 2006]. One consequence of this assumption is that it is possible to identify the components involved in each communication. Therefore, we consider that each message contains information about the component initiating the communication (sender) and the component receiving the communication (receiver). However, our framework can also accommodate other communication modes, such as broadcasting.

## 3.2 Case Study

Throughout this chapter, we consider a Travel Agency (TA) reservation system as a driving example. The reservation process is illustrated in the sequence diagram of Figure 3.1.

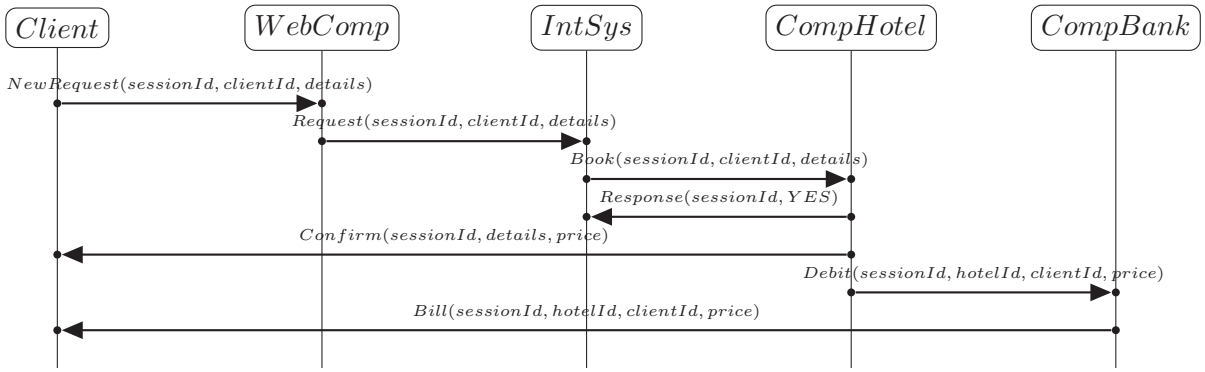



Figure 3.1: Accepted reservation scenario

First, the client requests a hotel reservation to a travel agency using a web component (*WebComp*) and informing his identifier and the reservation details. The travel agency internal system (*IntSys*) receives the request and tries to book a reservation by communicating with a hotel (*CompHotel*). If agreed, the requested hotel sends a confirmation to both the client and the travel agency. On the reservation date, the hotel charges the

client for the reservation by communicating with the bank (*CompBank*). Then, the client receives a debit confirmation from the bank.

The client can also cancel a reservation before the date of debit using *WebComp* indicating the identifier of the session. This scenario is represented by the sequence diagram of Figure 3.2.

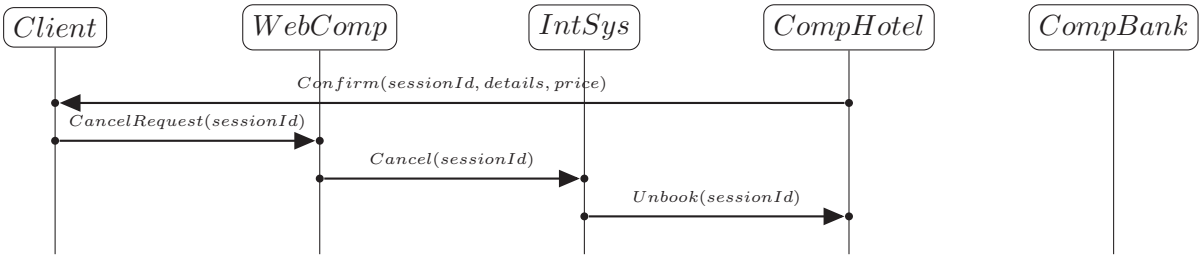



Figure 3.2: Reservation canceled scenario

In this case study we are interested in specifying the B2B contract between the travel agency and the various hotels that should specify theirs liabilities in case of damages caused to their clients.

## 3.3 Specifying Logs

In this section we define the models used to specify the logs: their content and how they are distributed.

### 3.3.1 API of Components

The contract includes a description of the components (and the information exchanged between them) that should be logged. We propose a model to represent the communications based on the API of the components. The API of a component define the services associated to the component and its communications with other components. In practice, the API may include different types of communications such as method calls or email messages.

The machine *ComponentsAPI* (Figure 3.3) specifies the set of components (set *COMP*) and the set of actions (set *ACTION*) forming the API of the components. The ellipsis marks ("...") in machines indicate information that should be completed.

MACHINE $ComponentsAPI$
SETS $COMP = \{\cdots\}$; $ACTION = \{\cdots\}$; $PARAM$
CONSTANTS $Interface$, $Invoke$, $NumParams$
PROPERTIES

`/* component that offers the function */`
$Interface \in ACTION \rightarrow COMP \wedge$
$Interface = \{\cdots\} \wedge$

`/* components that may invoke the function */`
$Invoke \in ACTION \rightarrow \mathcal{F}(COMP) \wedge$
$Invoke = \{\cdots\} \wedge$

`/* number of parameters */`
$NumParams \in ACTION \rightarrow \mathcal{N} \wedge$
$NumParams = \{\cdots\}$

END

Figure 3.3: Machine $ComponentsAPI$

The constant $Interface$ maps each action onto the component that performs the action. The constant $Invoke$ maps each action onto the set of component that may invoke the action. Internal actions of components may be represented using the same component to perform and invoke the action.

The parameters represent the information exchanged during communications and are specified using the set $PARAM$. Parameters can be specified by a complex description including their type and default values. We use here an abstract definition of parameter values based on a fixed given set. In practice, the set $PARAM$ works as a serialized representation of parameters. In machines, particular parameter values can be specified as constants of the type $PARAM$. For example, to specify a client's identifier we proceed as follows:

CONSTANT *clientId*
PROPERTIES $clientId \in PARAM$

The constant $NumParams$ maps each action onto the number of parameters exchanged in the action. For example, when requesting a reservation, the client provides two parameters: his identifier and the reservation requirements (location/date/number of nights).

In our model, each action has a unique identifier. To avoid any ambiguities we adopt the dot notation used in programming languages. For example, if a component *Comp* offers an action *Open* we name the action "*Comp.Open*". When no ambiguity can arise, the component name can be omitted.

**Example 3.1.** API of components in the Travel Agency system

To provide an example of instance of $ComponentsAPI$ using our case study, we complete the information of this machine with the following values:

$$
\begin{aligned}
&COMP = \{Client, WebComp, IntSys, CompHotel, CompBank\}\\
&ACTION = \{NewRequest, Request, Book, Response, Confirm,\\
&\qquad Debit, Bill, CancelRequest, Cancel, Unbook\}\\
&Interface = \{(NewRequest \mapsto WebComp), (Request \mapsto IntSys), (Book \mapsto CompHotel),\\
&\qquad (Response \mapsto IntSys), (Confirm \mapsto Client), (Debit \mapsto CompBank),\\
&\qquad (Bill \mapsto Client), (CancelRequest \mapsto WebComp), (Cancel \mapsto IntSys)\}\\
&\qquad (Unbook \mapsto CompHotel)\}\\
&Invoke = \{(NewRequest \mapsto \{Client\}), (Request \mapsto \{WebComp\}), (Book \mapsto \{IntSys\}),\\
&\qquad (Response \mapsto \{CompHotel\}), (Confirm \mapsto \{CompHotel\}),\\
&\qquad (Debit \mapsto \{CompHotel\}), (Bill \mapsto \{CompBank\}),\\
&\qquad (CancelRequest \mapsto \{Client\}), (Cancel \mapsto \{WebComp\}),\\
&\qquad (Unbook \mapsto \{IntSys\})\}\\
&NumParams = \{(NewRequest \mapsto 3), (Request \mapsto 3), (Book \mapsto 3),\\
&\qquad (Response \mapsto 2), (Confirm \mapsto 3), (Debit \mapsto 4),\\
&\qquad (Bill \mapsto 4), (CancelRequest \mapsto 1), (Cancel \mapsto 1), (Unbook \mapsto 1)\}
\end{aligned}
$$

Machine $ComponentsAPI$ only specifies the components and actions that will be logged. In some cases, certain actions and parameters cannot be recorded due to technical or privacy reasons [Garg et al. 2011]. In other cases, depending of the liabilities specified in the contract, it may not be necessary to specify the API of a given component. For example, a component used to communicate with an internal database that registers all reservations may not be included in the model because it does not appear in the definition of liabilities or its producer is a third party that does not take part in the contract.

The model can also include components that do not produce logs but which may appear in the messages. For example, this is the case of *Client* in our case study because it communicates with $WebComp$ and $CompBank$ but it is not a component of the system involved in the generation of the logs.

### 3.3.2 Log Files

We now define a model to specify the content of log files and their entries. Each *log entry* consists of:

- the type of entry: either *Send* if the communication is initiated by the action or *Rec* if the communication is received;
- the components that initiate (sender) and receive (receiver) the communication;

- the action and its associated parameters.

In most papers about trace analysis (Section 1.6), log files are represented by sequences of log entries. Because we want to be able to deal with distributed systems, we propose a extended notion of log file consisting not only of a sequence of entries but also of the set of components that are logged. In our approach, a log file is as a pair consisting of:

- the components that have their communications recorded in the log; in the text we refer to them as the *components* of a given log file.
- a sequence of log entries; in the text we refer to this sequence as the *content* of the log.

Machine *LogFiles* (Figure 3.4) specifies log entries (constant $ENTRY$) and log files (constant $LOG\_FILE$).

MACHINE *LogFiles*
SEES *ComponentsAPI*
SETS $TYPE = \{Send, Rec\}$
CONSTANTS $ENTRY, LOG\_FILE$
PROPERTIES

`/* log entries */`
$ENTRY = \{(tp, c_S, c_R, ac, par) \mid tp \in TYPE \wedge c_S \in COMP \wedge c_R \in COMP \wedge$
$ac \in ACTION \wedge par \in \mathsf{seq}(PARAM) \wedge \mathsf{size}(par) = NumParams(ac) \wedge$
$c_S \in Invoke(ac) \wedge c_R = Interface(ac)\} \wedge$

`/* log files */`
$LOG\_FILE = \{(comps, cont) \mid comps \in \mathcal{F}(COMP) \wedge cont \in \mathsf{iseq}(ENTRY) \wedge$
$(c1)$ $\forall(tp, c_S, c_R, ac, par).((tp, c_S, c_R, ac, par) \in \mathsf{ran}(cont) \Rightarrow$
$((tp = Send \wedge c_S \in comps) \vee (tp = Rec \wedge c_R \in comps))) \wedge$
$(c2)$ $\forall(en_A, en_B, c_S, c_R, ac, par).(en_A \in \mathsf{ran}(cont) \wedge en_B \in \mathsf{ran}(cont) \wedge$
$en_A = (Send, c_S, c_R, ac, par) \wedge en_B = (Rec, c_S, c_R, ac, par) \Rightarrow$
$Pos(en_A, cont) < Pos(en_B, cont))\}$

END

Figure 3.4: Machine *LogFiles*

The definition of $ENTRY$ imposes the consistency between the sender, receiver and number of parameters and the API defined in $ComponentsAPI$. The definition of $LOG\_FILE$ requires that log files contain only entries related to (i.e. sent and received by) the components involved ($c1$) and for each pair of corresponding *Send* and *Rec* entries, the *Send*

entry precedes the *Rec* entry ($c2$). *Pos* is a function (not defined here) that returns the position of an element in a sequence. We use iseq to specify that the log contains only unique entries (Section 3.1).

**Example 3.2.** Log file example

Let us consider the following log entries corresponding to a reservation request between the client and $IntSys$:

$en_1 \in ENTRY \wedge en_2 \in ENTRY \wedge en_3 \in ENTRY \wedge$
$en_1 = (Rec, Client, WebComp, NewRequest, [sessionId, clientId, details]) \wedge$
$en_2 = (Send, WebComp, IntSys, Request, [sessionId, clientId, details]) \wedge$
$en_3 = (Rec, WebComp, IntSys, Request, [sessionId, clientId, details])$

where the parameters $sessionId$, $clientId$ and $details$ correspond respectively to the session identifier, the client identifier and the reservation requirements. If components $WebComp$ and $IntSys$ are recorded in a single log file then we can represent a log file value as follow:

$log \in LOG\_FILE \wedge log = (\{WebComp, IntSys\}, [en_1, en_2, en_3])$

The values assigned to a log file must meet the requirements into the definition of $ENTRY$ and $LOG\_FILE$. The verification consists in establishing the proof obligation $log \in LOG\_FILE$, which includes conditions ($c1$) and ($c2$).

### 3.3.3 Logs Distribution

Contracts must specify how logs are distributed (Section 2.2). The notion of *log distribution* is generally not included in the formalisms proposed to define contract (Section 1.4). However, as mentioned in Section 1.5, to increase the probative value of the logs as digital evidence, it is recommended to provide a detailed description of the log infrastructure and the log management policy, including the process for generating and distributing logs.

Machine *LogDistribution* (Figure 3.5) defines a log distribution as a constant *Dist* that must be instantiated to include the set of components which are logged in the same log file.

MACHINE *LogDistribution*
SEES *ComponentsAPI*
CONSTANTS *Dist*
PROPERTIES

`/* log distribution */`
$Dist \in \mathcal{F}(\mathcal{F}(COMP)) \wedge$
$Dist = \{\dots\}$

END

Figure 3.5: Machine *LogDistribution*

The log distribution does not impose any constraints: components may be logged more than once or not logged at all. In Chapter 4, we analyze how the choice of the log distribution may affect the legal value of the digital evidence provided by the logs.

**Example 3.3.** Log distribution example

Let use consider three options for the distribution of logs of the Travel Agency system. The first option consists of two log files containing respectively the entries of $WebComp$ and $IntSys$. The second one involves a single log file. In the third one, the entries related to $WebComp$ are recorded twice. These distributions are represented respectively as follows:

- $Dist_1 = \{\{WebComp\}, \{IntSys\}\}$
- $Dist_2 = \{\{WebComp, IntSys\}\}$
- $Dist_3 = \{\{WebComp\}, \{WebComp, IntSys\}\}$

### 3.3.4 B Machines and Technical Annexes

The machine *ComponentsAPI* and *LogFiles* are included in Annex A to specify the information recorded in the logs. An implementation of machine *LogFiles* can be used to verify if a given value of a log file respects the definition imposed in $LOG\_FILE$, i.e., the entries are consistent with the component API and the ordering constraint.

Machine *LogDistribution* should be instantiated and included in Annex B to describe how log files are distributed.

The following table summarizes the information about the machines in the technical annexes:

| Machine | Annex | Information described | Instantiation required |
|---|---|---|---|
| *ComponentsAPI* | A | Information to be logged | Yes |
| *LogFiles* | A | Content of the log files | No |
| *LogDistribution* | B | Log distribution | Yes |

## 3.4 Operations on Distributed Logs

It may be necessary to manipulate different log files in order to obtain a single log file with entries relative to a given set of components. We define two functions for the manipulation of logs: extraction and merge. These functions are based on the theory of trace analysis and the well known relation *happened-before* introduced in the early work of Lamport [Lamport 1978].

Machine *LogOperations* (Figure 3.6) specifies the operations *Extract* and *Merge*. Their properties are defined in the following sections. Additionally, this machine specifies the constant $LOG\_SET$ representing finite sets of log files.

MACHINE *LogOperations*
SEES *ComponentsAPI*,*LogFiles*
CONSTANTS *LOG_SET*, *Extract*, *Merge*
PROPERTIES

`/* set of logs */`
$LOG\_SET = \mathcal{F}(LOG\_FILE)$

`/* Extract definition (Section 3.4.1) */`
`/* Merge definition (Section 3.4.2) */`
END

Figure 3.6: Machine *LogOperations*

### 3.4.1 Log Extraction

The function *Extract* allows us to extract a sub-log of a log file containing only the entries related to a given group of components.

**Definition 1.** Function *Extract*
The partial function $Extract \in (\mathcal{F}(COMP) \times LOG\_FILE) \nrightarrow LOG\_FILE$ maps every pair $(comps_{ext}, log)$, such that $log = (comps, cont)$ and $comps_{ext} \subseteq comps$, to $log_{ext} = (comps_{ext}, cont_{ext})$ being characterized by the following properties:

1. `/* extracted log content */`
   $\mathsf{ran}(cont_{ext}) = \{(tp, c_S, c_R, ac, par) \mid (tp, c_S, c_R, ac, par) \in \mathsf{ran}(cont) \wedge$
   $((tp = Send \wedge c_S \in comps_{ext}) \vee (tp = Rec \wedge c_R \in comps_{ext}))\}$
2. `/* preservation of entries order */`
   $\forall(en_A, en_B).(en_A \in \mathsf{ran}(cont_{ext}) \wedge en_B \in \mathsf{ran}(cont_{ext}) \wedge$
   $Pos(en_A, cont_{ext}) < Pos(en_B, cont_{ext}) \Rightarrow Pos(en_A, cont) < Pos(en_B, cont))$

The extracted log file ($log_{ext}$) contains all entries of *log* that are sent or received by components in $comps_{ext}$ (1.) and it respects the initial order of entries in *log* (2.).

**Example 3.4.** Example of *Extract*
Let us consider the following log file with entries related to **WebComp** and **IntSys**:

$log = (\{\mathbf{WebComp}, \mathbf{IntSys}\},$
$[(\mathbf{Rec}, Client, \mathbf{WebComp}, NewRequest, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{WebComp}, IntSys, Request, [sessionId, clientId, details]),$
$(\mathbf{Rec}, WebComp, \mathbf{IntSys}, Request, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{IntSys}, CompHotel, Book, [sessionId, clientId, details]),$
$(\mathbf{Rec}, CompHotel, \mathbf{IntSys}, Response, [sessionId, YES])])$

We can use *Extract* to obtain a log file related to *IntSys*:

$Extract(\{\mathbf{IntSys}\}, log) = (\{\mathbf{IntSys}\},$
$[(\mathbf{Rec}, WebComp, \mathbf{IntSys}, Request, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{IntSys}, CompHotel, Book, [sessionId, clientId, details]),$
$(\mathbf{Rec}, CompHotel, \mathbf{IntSys}, Response, [sessionId, YES])])$

### 3.4.2 Log Merging

The relation $Merge$ computes the *scenarios* consisting of all possible total orders of the entries of a given set of log files. More precisely, $Merge$ defines all the permutations of log entries, which respect the local order of each log file and the causal order between *Send* and *Rec* entries. This relation is similar to the operation used by [Hallal et al. 2006] (Section 1.6.10) to produce the scenarios of distributed traces.

**Definition 2.** Relation $Merge$
The relation $Merge \in LOG\_SET \leftrightarrow LOG\_FILE$ is defined for any pair $(LogSet, scenario) \in Merge$, such that $scenario = (comps_{scen}, cont_{scen})$ and:

1. `/* scenario components */`
   $comps_{scen} = \mathsf{union}(\{comps \mid \exists(cont).((comps, cont) \in LogSet)\})$

2. `/* scenario content */`
   $\mathsf{ran}(cont_{scen}) = \mathsf{union}(\{\mathsf{ran}(cont) \mid \exists(comps).((comps, cont) \in LogSet)\})$

3. `/* preservation of entries order */`
   $\forall comps, cont.((comps, cont) \in LogSet \Rightarrow Extract(comps, scenario) = (comps, cont))$

The last property states that the order of entries in *scenario* respects the order of entries of the log files in $LogSet$. By definition, the range of the relation $Merge$ only contains log files, such that the *Send* entries occur before the corresponding *Rec* entries ($\mathsf{ran}(Merge) \subseteq LOG\_FILE$). Therefore, this relation may result in an empty set if it is impossible to compute a scenario that respect the order of entries.

**Example 3.5.** Example of $Merge$

Let us consider two log files, $log_{WebComp}$ and $log_{IntSys}$, with the following content:

$log_{WebComp} = (\{\mathbf{WebComp}\},$
$[(\mathbf{Send}, \mathbf{WebComp}, IntSys, Request, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{WebComp}, IntSys, Cancel, [sessionId])])$

$log_{IntSys} = (\{\mathbf{IntSys}\},$
$[(\mathbf{Rec}, WebComp, \mathbf{IntSys}, Request, [sessionId, clientId, details])])$

If $LogSet = \{log_{WebComp}, log_{IntSys}\}$ the operation $Merge[\{LogSet\}]$ produces the set

$\{scen_1, scen_2\}$ such that:

$scen_1 =$ $(\{\mathbf{WebComp}, \mathbf{IntSys}\},$
$[(\mathbf{Send}, \mathbf{WebComp}, IntSys, Request, [sessionId, clientId, details]),$
$(\mathbf{Rec}, WebComp, \mathbf{IntSys}, \underline{Request}, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{WebComp}, IntSys, \underline{Cancel}, [sessionId]), ])$

$scen_2 =$ $(\{\mathbf{WebComp}, \mathbf{IntSys}\},$
$[(\mathbf{Send}, \mathbf{WebComp}, IntSys, Request, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{WebComp}, IntSys, \underline{Cancel}, [sessionId]),$
$(\mathbf{Rec}, WebComp, \mathbf{IntSys}, \underline{Request}, [sessionId, clientId, details])])$

The action *Cancel* can be permuted with the reception of *Request* since there is no ordering constraint between them.

## 3.5 Specifying and Verifying Log Properties

In this section we define the models used to describe and verify properties on log files.

### 3.5.1 Log Property

We use logical properties (called *log properties*) to describe the behaviors of the system. The various languages of properties described in Section 1.6 express properties in terms of conditions on sequences of log entries. However, this approach is not sufficient here because we want to express local properties on the logs attached to a given set of components. Therefore, we need not only to describe the conditions on log entries, but also the part of the system to which the property applies. To address this need we attach to each property the set of components of interest.

As an illustration, consider a property stating that "*IntSys* has sent a reservation request (*Book*) to the hotel *immediately after* (i.e. with no other communications in between) it received a request (*Request*)". This property holds for the following log file:

$log_1 = (\{\mathbf{IntSys}\},$
$[(\mathbf{Rec}, WebComp, \mathbf{IntSys}, \underline{Request}, [sessionId, clientId, details]),$
$(\mathbf{Send}, \mathbf{IntSys}, CompHotel, \underline{Book}, [sessionId, clientId, details])])$

However, the same property would not hold for the following log file that also includes *WebComp*'s actions:

$log_2 = (\{WebComp, \mathbf{IntSys}\},$
$[(Send, Client, WebComp, NewRequest, [sessionId, clientId, details]),$
$(Send, WebComp, IntSys, Request, [sessionId, clientId, details]),$
$(\mathbf{Rec}, WebComp, \mathbf{IntSys}, \underline{Request}, [sessionId, clientId, details]),$

$(Send, WebComp, CompHotel, Cancel, [sessionId])$,
$(\mathbf{Send}, \mathbf{IntSys}, CompHotel, \underline{Book}, [sessionId, clientId, details])])$

In our approach, we can specify that we are interested in a property related to *IntSys* only because we only want to check that *IntSys* has not communicated between *Request* and *Book*. The property should thus hold for both log files.

With this extended definition of log property it is possible to verify properties when not all logs are available. Similarly to the work on trace analysis proposed in [Arasteh et al. 2007] (Section 1.6.11).

Machine *LogProperties* (Figure 3.7) provides a formal definition of log properties (constant *PROP*). A log property is a pair $(comps, pred)$ consisting respectively of:

- the components concerned by the property. In the text, we refer to this set as the *components* of the property;
- a function mapping a log file into a boolean value indicating if the property holds for a given log. In the text, we refer to this function as the *predicate* of the property.

MACHINE *LogProperties*
SEES *ComponentsAPI*, *LogFiles*
CONSTANTS *PROP*, *PropComps*, *PropPredicate*
PROPERTIES

`/* log properties */`
$PROP = \{(comps, pred) \mid comps \in \mathcal{F}(COMP) \land pred \in (LOG\_FILE \nrightarrow \mathsf{BOOL}) \land$
$(c1)$ $\forall(log).(log \in \mathsf{dom}(pred) \Rightarrow \exists(comps_{log}, cont_{log}).(log = (comps_{log}, cont_{log}) \land$
$comps_{log} = comps))\} \land$

`/* selectors for log properties */`
$PropComps \in PROP \rightarrow \mathcal{F}(COMP) \land$
$PropPredicate \in PROP \rightarrow (LOG\_FILE \nrightarrow \mathsf{BOOL}) \land$
$(c2)$ $\forall(comps, pred).((comps, pred) \in PROP \Rightarrow$
$PropComps(comps, pred) = comps \land PropPredicate(comps, pred) = pred)$

END

Figure 3.7: Machine *LogProperties*

The definition of *PROP* requires that the property predicate is a partial function such that its domain is defined only for log files related to the components of interest ($c1$). The selectors *PropComps* and *PropPredicate* return each part of a given log property ($c2$) and are used to define the values of log properties, as illustrated in the following example.

**Example 3.6.** Log property example

We specify the property mentioned at the beginning of this section stating that $IntSys$ sends a reservation message immediately after it has received a request. First we specify an identifier and the components attached to the property:
$prop_{Demand} \in PROP \wedge PropComps(prop_{Demand}) = \{IntSys\}$

Assuming that $sessionId$, $clientId$ and $details$ have been defined previously, we specify the predicate of the property using the lambda notation:

$PropPredicate(prop_{Demand}) =$
$\lambda(log).(log \in LOG\_FILE \wedge \exists(cont).(log = (\{IntSys\}, cont)) \mid \mathsf{bool}(\exists(cont, en_A, en_B).$
$(log = (\{IntSys\}, cont) \wedge en_A \in \mathsf{ran}(cont) \wedge en_B \in \mathsf{ran}(cont) \wedge$
$en_A = (Rec, WebComp, IntSys, Request, [sessionId, clientId, details]) \wedge$
$en_B = (Send, IntSys, CompHotel, Book, [sessionId, clientId, details]) \wedge$
$Pos(en_B, cont) = Pos(en_A, cont) + 1)))$

This property applies to a given session and client. We see in the following section how to provide a generalization of it based on a set of parameters. Similarly to the definition of log files, it is also possible to check if a given property $prop$ is consistent with the definition of log property by verifying the proof obligation $prop \in PROP$.

We can define an operation that takes as input a log file ($log$) and a log property ($prop$) and return a boolean ($hold$) indicating if the property holds or not for the log:

$hold \leftarrow VerifyProperty(log, prop) \hat{=}$
PRE
$log \in LOG\_FILE \wedge prop \in PROP \wedge$
$\exists(comps, cont).((comps, cont) = log \wedge PropComps(prop) \subseteq comps)$
THEN
$hold := (PropPredicate(prop))(Extract(PropComps(prop), log))$
END

The pre-condition ensures that the components related to the log file contain at least the components attached to the log property. Then, $Extract$ is applied to ensure that only the concerned entries are used in the analysis of the property.

This definition however does not take into account distributed logs and it is not included in the technical annexes. In Section 3.5.3 we provide a version of this operation to verify log properties that also takes into account distributed logs, which constitutes the log analyzer.

### 3.5.2 Parametric Properties

It is often useful to generalize properties with respect to certain parameters [Chen & Roşu 2009]. Many proposals in traces analysis (Section 1.6) and contract formalisms (Section 1.4) provide a way to characterize *parametric properties*. For example,

one can specify a property that, given a client identifier, verifies if a reservation has been done for this client. We introduce now a model to specify parametric properties.

Machine *ParametricProperties* (Figure 3.8) defines parametric properties (constant $PAR\_PROP$) consisting of a set of components and a partial function that maps a sequence of parameters to a log property.

MACHINE *ParametricProperties*
SEES *ComponentsAPI*, *LogProperties*
CONSTANTS $PAR\_PROP$, $PPropComps$, $PPropPredicate$
PROPERTIES

`/* parametric properties */`
$PAR\_PROP = \{(comps, ppred) \mid comps \in \mathcal{F}(COMP) \land ppred \in \mathsf{seq}(PARAM) \nrightarrow PROP \land$
$\forall(prop).(prop \in \mathsf{ran}(ppred) \Rightarrow comps = PropComps(prop))\} \land$

`/* selectors for parametric properties */`
$PPropComps \in PAR\_PROP \rightarrow \mathcal{F}(COMP) \land$
$PPropPredicate \in PAR\_PROP \rightarrow (\mathsf{seq}(PARAM) \nrightarrow PROP) \land$
$\forall(comps, ppred).((comps, ppred) \in PAR\_PROP \Rightarrow$
$PPropComps(comps, pred) = comps \land PPropPredicate(comps, pred) = pred)$

END

Figure 3.8: Machine *ParametricProperties*

The definition of $PAR\_PROP$ states that the instantiated properties have the same components attached to the parametric properties. Similarly to log properties we define the selectors $PPropComps$ and $PPropPredicate$ that return each part of a parametric property. The domain of the predicate of a parametric property should be defined only for the number of parameters accepted by the property, as illustrated by the following example.

**Example 3.7.** Parametric property example

We specify now a parametric version of the log property $prop_{Demand}$ of Example 3.6 that takes as parameters the session identifier, client identifiers and reservation details. First, we should define the components attached to the parametric property:

$parProp_{Demand} \in PAR\_PROP \land$
$PPropComps(parProp_{Demand}) = \{IntSys\} \land$

Then, we define its predicate using the lambda notation:

$PPropPredicate(parProp_{Demand}) = \lambda(par).(par \in \mathsf{seq}(PARAM) \wedge \mathsf{size}(par) = 3 \mid \{IntSys\} \mapsto$
$\lambda(log).(log \in LOG\_FILE \wedge \exists(cont).(log = (\{IntSys\}, cont)) \mid \mathsf{bool}(\exists(cont, en_A, en_B).$
$(log = (\{IntSys\}, cont) \wedge en_A \in \mathsf{ran}(cont) \wedge en_B \in \mathsf{ran}(cont) \wedge$
$en_A = (Rec, WebComp, IntSys, Request, [par(1), par(2), par(3)]) \wedge$
$en_B = (Send, IntSys, CompHotel, Book, [par(1), par(2), par(3)]) \wedge$
$Pos(en_B, cont) = Pos(en_A, cont) + 1))))$

We use $par(1), par(2)$ and $par(3)$ to refer respectively to the session identifier, client identifier and reservation details. In order to obtain an instance of log property for given parameters *sessionId*, *clientId* and *details*, we use the predicate of the parametric property:

$prop_{Demand} \in PROP \wedge$
$prop_{Demand} = (PPropPredicate(parProp_{Demand}))([sessionId, clientId, details])$

### 3.5.3 Analysis of Distributed Logs

We present now the specification of the log analyzer which evaluates a given log property for a set of distributed logs. The analysis of distributed logs is based on the scenarios obtained by merging the logs, similarly to the approaches of trace analysis proposed in [Arasteh et al. 2007, Hallal et al. 2006] (Section 1.6.10).

Machine *LogAnalyzer* (Figure 3.9) defines the operation $VerifyProperty$ which takes as input a set of log files ($LogSet$) and a log property ($prop$).

MACHINE *LogAnalyzer*
SEES *LogOperations*, *LogProperties*
OPERATIONS
$scen, ok \leftarrow VerifyProperty(LogSet, prop) \hat{=}$
PRE
$LogSet \in LOG\_SET \wedge prop \in PROP \wedge$
(c1) $PropComps(prop) \subseteq \mathsf{union}(\{comps \mid \exists(cont).((comps, cont) \in LogSet)\})$
THEN
LET $temp$ BE $temp = Extract[\{PropComps(prop)\} \times Merge[\{LogSet\}]]$ IN
$scen := temp \ ||$
$ok := temp \cap PropPredicate(prop)^{-1}[\{\mathsf{TRUE}\}]$
END
END
END

Figure 3.9: Operation $VerifyProperty$

To comply with the constraints of $PROP$ (Figure 3.7), the pre-condition states that the components of the log files must contain at least the components attached to the property ($c1$). We should also make sure that each log file in $LogSet$ respects the definition of $LOG\_FILE$, i.e. each log file only contains entries consistent with the API and the order of $Send/Rec$ entries.

The operation $VerifyProperty$ computes two results: $scen$, the set of all scenarios after extracting the entries related to the components attached to the property (computed in $temp$); and $ok$, the subset of these scenarios where the property holds. The ratio between these two results provide information on the validity of the researched property for the given logs:

- if $ok = scen$ the property holds for all scenarios.
- if $ok = \emptyset$ the property is false for all scenarios.

If none of these conditions holds then a further analysis of the scenarios can be conducted. One possibility is to increase the number of log files used in the analysis, which can have the effect of reducing the number of scenarios. However, it is possible that even using a larges set of logs it is still not possible to show a clear conclusion. In this case, a detailed analysis of each scenario by an expert may be necessary to assess the likelihood of each scenario and take into account contextual factors which are not included in the formal model.

Regarding the complexity of the operation, it is known that the problem of counting the scenarios of distributed logs is #P-complete (equivalent do count the number of solutions of a NP-complete problem), with respect to the number of entries in the logs. However, in [Brightwell & Winkler 1991], the authors show that the number of scenarios may also be estimated using randomized polynomial-time algorithms. Optimizations can also be be made for properties stating conditions specifically about the order of the entries [Ivanov 2005]. These optimizations consists of trying to match the conditions of the property using graphs (represent by parallel posets) that represents the causality relation between the entries of the logs.

**Example 3.8.** Distributed log analysis example

We show how the log analyzer can be used to verify the property $prop_{LateCancel}$ stating

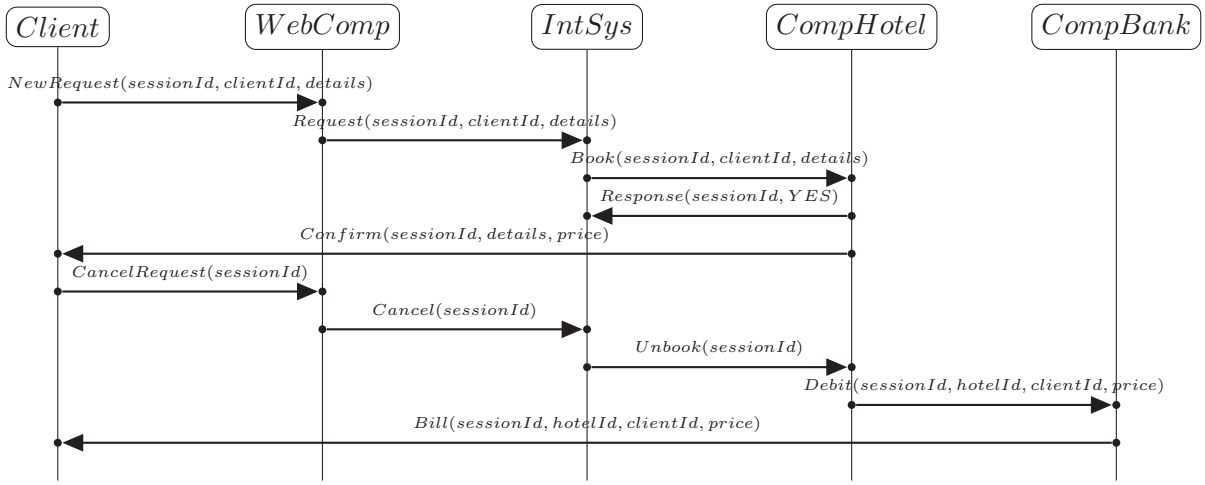
that the client is charged for a canceled reservation:

$prop_{LateCancel} \in PROP \wedge PropComps(prop_{LateCancel}) = \{WebComp, CompHotel\} \wedge$
$PropPredicate(prop_{LateCancel}) = \lambda(\log).(\log \in LOG\_FILE \wedge$
$\exists(cont).(\log = (\{WebComp, CompHotel\}, cont)) \mid \mathsf{bool}(\exists(cont, en_A, en_B).$
$(log = (\{WebComp, CompHotel\}, cont) \wedge en_A \in \mathsf{ran}(cont) \wedge en_B \in \mathsf{ran}(cont) \wedge$
$en_A = (Rec, Client, WebComp, CancelRequest, [sessionId]) \wedge$
$en_B = (Send, CompHotel, CompBank, Debit, [sessionId, hotelId, clientId, price]) \wedge$
$Pos(en_A, cont) < Pos(en_B, cont))))$

The predicate holds when *CancelRequest* occurs before *Debit*. For now, we assume that the values for *sessionId*, *hotelId*, *clientId* and *price* are supplied by the client and have been defined previously. Consider the following log distribution, where each component produces a log file:

$$Dist = \{\{WebComp\}, \{IntSys\}, \{CompHotel\}, \{CompBank\}\}$$

and the communications illustrated by the following diagram:




If we execute *VerifyProperty* with the logs of *WebComp* and *CompHotel*, then 126 scenarios are produced and the property holds for 105 of them. The high number of scenarios is a consequence of the absence of interaction between *WebComp* and *CompHotel*: as a consequence, there are no restrictions on the ordering of entries, except the local order of each log file. However, if we also use the log of *IntSys*, then only 10 scenarios are produced, and the property holds for all of them, which allows us to get to a positive conclusion.

### 3.5.4 B Machines and Technical Annexes

Machines *LogProperties* and *ParametricProperties* are part of Annex A and are used to specify failures, which will be discussed in the next chapter.

Machines *LogOperations* and *LogAnalyzer* provide the formal definition of the log analyzer and are included in Annex D. The procedure defining the use of the log analyzer to establish liabilities is the subject of Annex C and it is detailed in the next chapter.

The following table summarizes information about the machines in the technical annexes:

| Machine | Annex | Information described | Instantiation required |
|---|---|---|---|
| *LogProperties*<br>*ParametricProperties* | A | Properties of logs | No<br>(used in Chapter 4) |
| *LogOperations* | D | Distributed logs operations used by the log analyzer | No |
| *LogAnalyzer* | D | Log analyzer | No |

## 3.6 Incremental Analysis

At the time of analysis, some logs may not be available either due to legal reasons (availability depending on a formal demand) or technical reasons (such as log data that need to be decrypted). For example, in our study case the access to the log file of *CompBank* may not be immediate. In this case, the log analysis may take place using the logs of only a part of the system (Example 3.8). However, depending on the results it may be necessary to perform a new analysis using a larger set of logs that can help to approximate the real behavior. In this section, we propose an incremental version of the log analyzer.

We include in machine *LogAnalyzer* the operation $IncrVerifyProperty$ (Figure 3.10) that takes as input a set of log files ($LogSet$), a log property (*prop*) and the result *ok* from a previous log analysis.

$iscen, iok \leftarrow IncrVerifyProperty(LogSet, prop, ok) \mathrel{\hat{=}}$
PRE
$LogSet \in LOG\_SET \wedge prop \in PROP \wedge scen \in LOG\_SET \wedge ok \in LOG\_SET \wedge$
$ok \subseteq PropPredicate(prop)^{-1}[\{\mathsf{TRUE}\}]$
THEN
LET $temp$ BE $temp = Extract[\{PropComps(prop)\} \times Merge[\{LogSet\}]]$ IN
$iscen := temp \ ||$
$iok := temp \ \cap \ ok$
END
END

Figure 3.10: Operation $IncrVerifyProperty$

The computation of *iscen* are similar to the operations used in the previous version (Figure 3.9). However, the computation of *iok* consists of comparing the results of the scenarios produced (*temp*) with the scenarios for which the property was previously verified (*ok*). The main interest is to compute the result of *iok* without having to check again the property.

The correctness of the proposed operation is based on the following property:

**Property 3.1.** Let $LogSet, LogSet' \in LOG\_SET$ such that $LogSet \subseteq LogSet'$ then for any $comps \subseteq \mathsf{union}(\{comps \mid \exists(cont).((comps, cont) \in LogSet)\})$:

$$Extract[\{comps\} \times Merge[\{LogSet'\}]] \subseteq Extract[\{comps\} \times Merge[\{LogSet\}]]$$

This property says that the additional logs in $LogSet'$ can only restrain the scenarios that will be produced by *LogSet*, introducing more causalities. The *Extract* function ensures that we compare scenarios dedicated to the same set of components (they have the same entries). This property is based on an hypothesis we have made, imposing that events related to a given component *comp* are the same ones in all logs associated to this component, and in the same order. More formally:

$$\begin{array}{c} \forall log1, log2, comp \ . \ (log1 \in LOG\_FILE \wedge \ log2 \in LOG\_FILE \wedge comp \in COMP \\ \wedge \ (\{comp\}, log1) \in \ dom(extract) \wedge \ (\{comp\}, log2) \in \ dom(extract) \\ \Rightarrow Extract(\{comp\}, log1) \ = \ Extract(\{comp\}, log2) \end{array}$$

Property 3.1 allows us to compare the results of the incremental log analyzer to the results of the previous log analyzer in the following theorem.

**Theorem 3.1.** Result of incremental log analyzer

Let $prop \in PROP$ and $LogSet, LogSet' \in LOG\_SET$ such that $LogSet \subseteq LogSet'$ and:

$$\begin{aligned} scen, ok &\leftarrow VerifyProperty(LogSet, prop); \\ iscen, iok &\leftarrow IncrVerifyProperty(LogSet', prop, ok); \\ scen', ok' &\leftarrow VerifyProperty(LogSet', prop) \end{aligned}$$

Then we have $iscen = scen'$ and $iok = ok'$, i.e. the incremental analysis produces the same result as re-analyzing the property using more logs.

*Proof.* $iscen = scen'$ are equals because they are computed by an identical operation. To prove that $iok = ok$, we start with the definition of $iok$:

$$iok = iscen \cap ok$$

If we apply the definition of $ok$ and $iscen = scen'$:

$$iok = scen' \cap (PropPredicate^{-1}(prop)[\{\mathsf{TRUE}\}] \cap scen)$$

From Property 3.1 we know that $scen' \subseteq scen$ because the new logs used to compute $scen'$ can only add constraints on the scenarios of $scen$, then:

$$iok = scen' \cap PropPredicate^{-1}(prop)[\{\mathsf{TRUE}\}]$$

Which is precisely the definition of $ok'$. □

**Example 3.9.** Incremental analysis example

The log analysis described in Example 3.8 can be performed incrementally. The incremental analysis produces only 10 scenarios which are all evaluated to $ok$ and we may conclude that the property holds ($iscen = iok$).

In [Mazza et al. 2010], we propose a different version of $IncrVerifyProperty$ that takes as input the scenarios produced in the previous analysis ($scen$). The operation verifies if the additional logs contain entries that are eliminates some some scenarios in $scen$ in order to approximate the result of $iscen$ and $iok$. This approach is useful when the number of merged scenarios is large and the final result is small subset of them. For example, suppose that in the first analysis using $LogSet$ the merge produces 1000 scenarios and only 2 scenarios remained after the extraction ($\mathsf{card}(scen) = 2$). In the second analysis, using $LogSet' = LogSet \cup \{log_A\}$, this approach consists in only checking if the entries in $log_A$ eliminates one of the scenarios of $scen$, rather that compute the merging of $LogSet$' that may produce even more than 1000 scenarios. However, the approach proposed in [Mazza et al. 2010] only produces an approximation of the results because, due to extraction, we loose information about entries, consequently producing scenarios that would otherwise be eliminated using the additional logs.

Additional logs should help to reduce the number of scenarios by facing new causalities between. To choose which logs have to be added a possible heuristic is to analyze the scenarios in $scen$ in order to maximize the number of causality relation between $Send/Rec$ entries. For example, during the analysis of $prop_{LateCancel}$ (Example 3.8) the produced

scenarios contain 5 communications to/from *IntSys* and only one communication with *CompBank*, therefore adding *IntSys*'s log is more likely to reduce the scenarios than *CompBank*'s log.

## 3.7 Technical Annexes and Machines

We now summarize how machines can be used to define the content of technical annexes, as described in Chapter 2.

Annex A includes an instance of machine *ComponentsAPI* defining the information that is recorded in the logs and machine *LogFiles* that defines the content of logs.

Annex B includes an instance of machine *LogDistribution* describing how log files are distributed. In the next chapter, we specify desirable characteristics of log distributions to provide log files more likely to be accepted as digital evidence.

Annex D contains machines *LogOperations* and *LogAnalyzer* that provide the formal definition of the log analyzer. The details about the use of the log analyzer is described in the next chapter.

The following table summarizes the information about all machines presented in this chapter and the technical annexes:

| Machine | Annex | Information described | To be instantiated? |
|---|---|---|---|
| *ComponentsAPI* | A | Information to be logged | Yes |
| *LogFiles* | A | The content of the log files | No |
| *LogDistribution* | B | Log distribution | Yes |
| *LogProperties*<br>*ParametricProperties* | A | Properties of logs | No<br>(used in Chapter 4) |
| *LogOperations* | D | Distributed logs operations used by the log analyzer | No |
| *LogAnalyzer* | D | Log Analyzer | No |

## 3.8 Contributions of the Chapter

In this chapter we propose a set of formal models to specify the logs and their distribution. A first contribution, consists of precisely defining the set of constraints about the log files and their entries. These constraints are stated in machine *LogFiles* (Section 3.3.2) and can be used to verify integrity of a given log file.

The second contribution of our approach is that we take into account the analysis of distributed logs. Some contract formalisms assume distributed logs, e.g. POETS [Andersen et al. 2006] and SLAng [Lamanna et al. 2003]. However, these approaches

are usually based on the existence of a common clock for all components, and analysis is performed always with a single scenario. Since this is not always the case [Schwarz & Mattern 1994], our approach also take into account the analysis of distributed traces (Section 1.6) where communications may occur concurrently without established order.

An important contribution and a original aspect of our framework is that we propose a way to specify properties that are only concerned with a part of the system. An advantage of this approach is to make possible the analysis when not all logs are available, as seen in Example 3.8 (Section 3.5.3). Similar approaches have been offered to specify local properties, e.g. [Saleh et al. 2007, Vardi 2008] however these approaches do not enforce the use of the logs related to the properties in their analysis. Finally, this last aspect allow us to provide an incremental way to analyze log properties. The main advantage of this approach is the possibility to obtain a conclusive result, without having to reanalyze the value of the property.

# Chapter 4

# Specifying and Establishing Liabilities

The log analysis presented in the previous chapter allows us to extract useful information from the logs. The next step is to use this information to establish liabilities, which is the subject of this chapter.

## 4.1 LISE Approach

As mentioned in Chapter 2, our objective is to help in the elaboration of the parts of the contracts that are dedicated to software failures and the resulting liabilities in case of damages. These failures are notified in the form of *claims*, which represent a complaint from an actor against a party of the contract.

The contract specifies (Article 4, Section 2.2.4) that when a claim is notified the parties must initiate the process of collection and analysis of the logs, defined in Annex C and referred in the contract as the *log analysis procedure*. In practice, the contract includes a selection of the most significant claims for the parties. For example, in our case study the parties may decide to specify liabilities when the client claims that he has sent a reservation but has not received any confirmation. Article 4 of the contract also specifies that, if liabilities for a given failure are not explicitly defined, then the parties agree to resort to an expert that may use the logs, the log analyzer and any other resources to designate the liable parties.

The log analysis procedure (Figure 4.1) starts with a claim from the plaintiff against the defendant. The party identified in the contract as the leader of the investigation (called here the *analyst*) must get the logs necessary to analyze the claim (*Log Collect*). Ideally, it should always be possible to obtain the totality of the logs. However, depending on the claim and the availability of logs, the analyst may analyze the claim using only a subset of the logs.

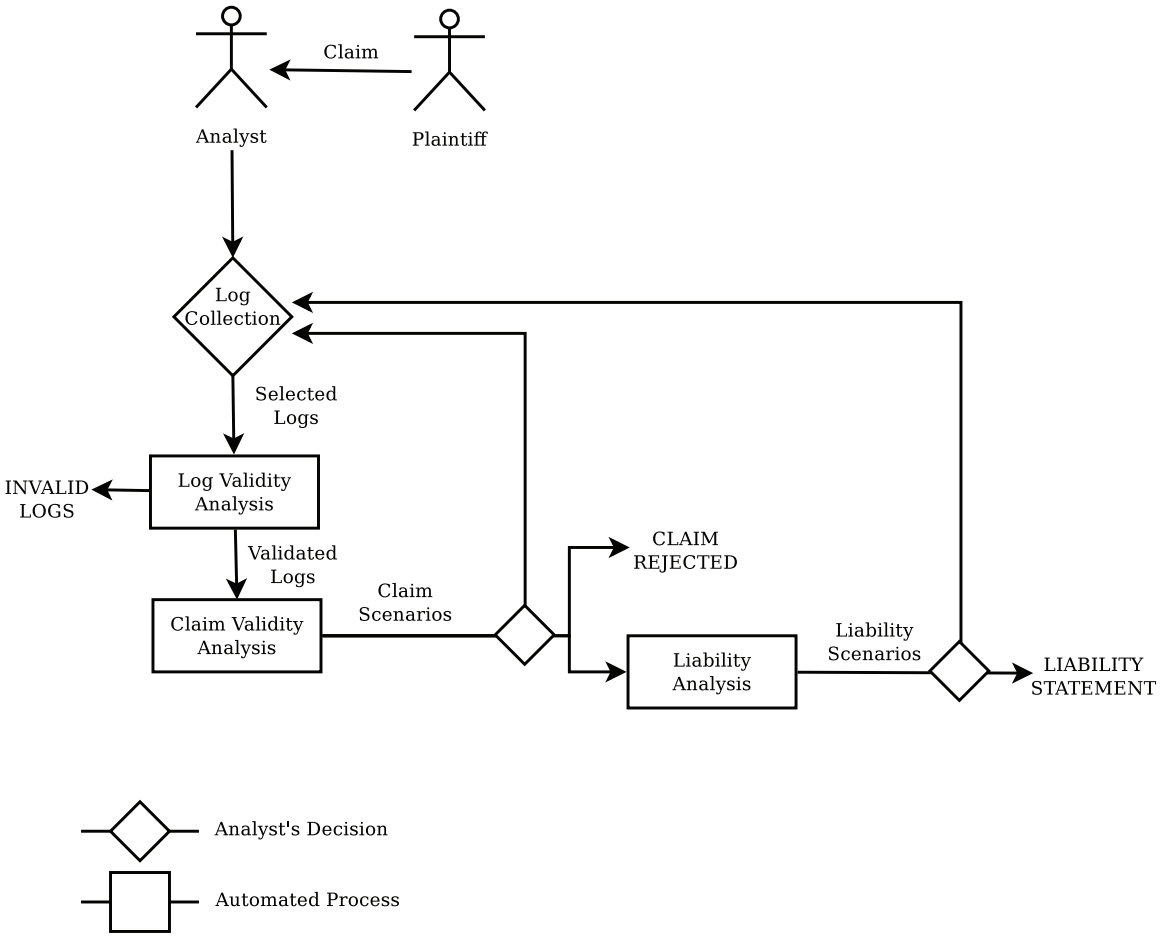



Figure 4.1: Log analysis procedure

After the collection of the logs, the analyst has to check if they can be accepted as digital evidence (*Log Validity Analysis*). For example, the logs must not present signs of tampering (Section 1.5.1). Certain integrity properties of the logs may be verified automatically (see Chapter 5) based on the formal definitions in $LOG\_FILE$ and $ENTRY$ (Section 3.3.2).

When the validity of the logs is established, the log analyzer (defined in Section 3.5.3) is applied to check the property that represents the claim (*Claim Validity Analysis*). The result of this analysis is a pair of sets of *Claim Scenarios* for which the claim is accepted and rejected respectively (as seen in Example 3.8). However, as explained in Section 3.6, if the results are not precise enough it may be necessary to perform a further analysis including additional logs. In this case, the analyst may iterate the procedure and return to the log collection step.

If the claim is accepted, then the liability allocation process (*Liability Analysis*) takes place. This process consists in determining the component (or set of components) that did not execute correctly (erroneous components). The liability relation between errors and parties are specified in the tables of Annex E of the contract (Section 2.2.3). Each Claim Scenario resulting from the Claim Validity Analysis is verified in order to determine the erroneous components. However, because the analysis can be performed with only a subset of the logs, its results may not be precise enough to establish liabilities. In this case, the analyst may return to the log collection step in order to attempt to reduce the number of scenarios (Section 3.6).

First, we introduce the models to represent claims and liabilities (Section 4.2). Then, we define the procedure to establish liabilities (Section 4.3) and we propose formal criteria on log distributions to increase the chances that logs will be accepted as digital evidence (Section 4.4). Finally, we summarize the main contributions of this chapter (Section 4.5).

## 4.2 Specifying Liabilities

In order to define our models for claims (Section 4.2.2) and liabilities (Section 4.2.3) we must first specify the parties involved (Section 4.2.1).

### 4.2.1 Parties

The formalisms described in Section 1.4 usually do not address the legal implications of third parties, such as customers (even when they support blame assignment [Xu et al. 2005, Hvitved 2010]). In the contract model proposed in Section 2.2, we distinguish between two types of legal entities:

- *Signing parties* (or simply *parties*) – the parties that elaborate and sign the contract (e.g. software providers and integrators).
- *Third parties* – the entities that are not involved in the elaboration of the contract (e.g. costumers). Third parties still have the right to legally sue the signing parties.

Every signing party is identified by a unique name. In contrast, some third parties are referred in the contract only by the role that they play in the system. As an illustration, consider our case study where the travel agency "Thomas Cook" and the hotel "iBis" are the signing parties and theirs customers are third parties mentioned in the contract simply as "Client".

Machine *ContractAgents* (Figure 4.2) should be completed to specify the set of all legal entities (called *agents*) mentioned in the contract (set $AGENT$). Subsets $SIGN\_PARTY$ and $THIRD\_PARTY$ should contain respectively the signing parties and the third parties.

MACHINE *ContractAgents*
SETS
$AGENT = \{\dots\}$
CONSTANTS $SIGN\_PARTY$,$THIRD\_PARTY$
PROPERTIES

$SIGN\_PARTY \subseteq AGENT \wedge$
$THIRD\_PARTY \subseteq AGENT \wedge$
$SIGN\_PARTY = \{\dots\} \wedge THIRD\_PARTY = \{\dots\} \wedge$
$SIGN\_PARTY \cup THIRD\_PARTY = AGENT \wedge$
$SIGN\_PARTY \cap THIRD\_PARTY = \emptyset$

END

Figure 4.2: Machine *ContractAgents*

**Example 4.1.** Parties

We provide an example of instance of the machine *ContractAgents* with the following values of our case study:

$AGENT = \{Client, ThomasCook, iBis, BNP\}$
$SIGN\_PARTY = \{ThomasCook, iBis\}$
$THIRD\_PARTY = \{Client, BNP\}$

Note that only signing parties can be made liable in the contract (a legal contract cannot impose any commitments on third parties). In our case study, we will consider that Thomas Cook is liable for the components $WebComp$ and $IntSys$, and iBis is liable for the component $CompHotel$. The component $CompBank$ is under the responsibility of the third party BNP.

### 4.2.2 Claims

As mentioned at the beginning of this chapter, the notification of failures takes the form of claims, which represent legal complaints from a plaintiff against a defendant. Each claim is formally described as a plaintiff, a defendant and a parametric property representing the alleged failure that motivates the claim.

Since claims can occur several times in different contexts, we represent failures using parametric properties and we define a specific instance whenever a party initiates a claim. A *claim instance* consists of a claim with a sequence of parameter values.

Machine *Claims* (Figure 4.3) should be instantiated to define the set of identified claims. It specifies the set of claims (constant $CLAIM$), each one consisting of the plaintiff, the defendant and the parametric property describing the associated failure. The machine also defines the set of claim instances (constant $CLAIM\_INST$).

MACHINE *Claims*
SEES *LogFiles*, *ParametricProperties*, *ContractAgents*
CONSTANTS $CLAIM$, $CLAIM\_INST$, $DECLARED\_CLAIMS$, ...
PROPERTIES

```
/* claims */
```
$CLAIM = AGENT \times SIGN\_PARTY \times PAR\_PROP \wedge$

```
/* claim instances */
```
$CLAIM\_INST = \{(claim, par) \mid claim \in CLAIM \wedge par \in \mathsf{seq}(PARAM) \wedge$
(c1) $\quad \exists(plain, def, parProp).(claim = (plain, def, parProp) \wedge$
$\forall par_{claim}.(par_{claim} \in \mathsf{dom}(PPropPredicate(parProp)) \Rightarrow \mathsf{size}(par_{claim}) = \mathsf{size}(par)))\} \wedge$

```
/* property and claim declarations */
```
...

```
/* claims in contract */
```
$DECLARED\_CLAIMS \subseteq CLAIM \wedge$

$DECLARED\_CLAIMS = \{\dots\}$

END

Figure 4.3: Machine *Claims*

Property (*c*1) states that the number of parameters of the claim instance must be the same as the number of parameters of the property attached to the claim.

For a given contract, claims and properties attached to them could be declared as constants and specified in the PROPERTIES section. The set of declared claims must be defined in the constant $DECLARED\_CLAIMS$.

**Example 4.2.** Claim $claim_{NoRoom}$ example

Let us specify the claim where the client purports that he has sent a request but has not received any confirmation for the reservation. First, in the CONSTANT section we declare the name of the claim and the property attached to the claim:

MACHINE *Claims*
CONSTANTS ..., $claim_{NoRoom}, parProp_{NoRoom}$

Then, in the PROPERTIES section we specify $parProp_{NoRoom}$ that takes two parameters, the session identifier ($par(1)$) and the client identifier ($par(2)$):

$parProp_{NoRoom} \in PAR\_PROP \wedge$
$PPropComps(parProp_{NoRoom}) = \{\mathbf{WebComp}, \mathbf{CompHotel}\} \wedge$

$PPropPredicate(parProp_{NoRoom}) = \lambda(par).(\mathsf{size}(par) = 2 \mid \{WebComp, CompHotel\} \mapsto$
$\lambda(\log).(\log \in LOG\_FILE \wedge \exists(cont).(\log = (\{WebComp, CompHotel\}, cont)) \mid$
$\mathsf{bool}(\exists(cont, details, price).(log = (\{WebComp, CompHotel\}, cont) \wedge$
$details \in PARAM \wedge price \in PARAM \wedge$
$(\mathbf{Rec}, Client, \mathbf{WebComp}, NewRequest, [par(1), par(2), details]) \in \mathsf{ran}(cont) \wedge$
$(\mathbf{Send}, \mathbf{CompHotel}, Client, Confirm, [par(1), details, price]) \notin \mathsf{ran}(cont))))) \wedge$

The property holds if $WebComp$ receives the request ($NewRequest$), but $CompHotel$ does not send the confirmation ($Confirm$). The predicate is defined for any possible values of $details$ and $price$ ($\exists(details, price)$).

We can specify the claim as follows:

$claim_{NoRoom} \in CLAIM \wedge$
$claim_{NoRoom} = (Client, ThomasCook, parProp_{NoRoom}) \wedge$
$DECLARED\_CLAIMS = \{claim_{NoRoom}\}$

An instance of $claim_{NoRoom}$ for a client with identifier $clientId$ during a given session $sessionId$ is represented as follows:

$$(claim_{NoRoom}, [clientId, sessionId])$$

The parties may decide which failures must be included in the model based on a common agreement or adopt a more systematic approach using, for example, techniques of *failure mode, effects and criticality analysis* (FMECA) [Peláez & Bowles 1996, Chin et al. 2009] which consists in analyzing potential failures and classifying them by likelihood and severity. For example, in our case study, besides the failure $claim_{NoRoom}$, the parties can also decide to specify a claim where the client purports that he has been charged for a canceled reservation.

### 4.2.3 Liabilities

As explained in Section 2.2.4, contractual liabilities take the form of tables (one for each claim) associating erroneous components with liable parties. Each line of these tables characterizes an error in a component. As an illustration, the table in Section 2.2.4 specifies that if errors occurred in "the obstacle detection functionality due to extreme weather condition", then the system integrator and supplier should be liable for the failure. Here we are interested in representing only conditions that can be verified using the logs. To this aim, we provide in this section a way to relate errors, failures and the corresponding liable parties.

For example, suppose the parties wish to specify liabilities for the claim $claim_{NoRoom}$ (Example 4.2) for the errors defined in the table of Figure 4.4.

| **Annex E: Assignment of liabilities for claim** $claim_{NoRoom}$ | | |
|---|---|---|
| **If errors have occurred:** | **Then, the liabilities will be assigned to:** | |
| | Thomas Cook | iBis |
| $WebComp$ or $IntSys$ does not pass the request ($parProp_{NoForward}$) | × | |
| $CompHotel$ does not send confirmation ($parProp_{NoConfirm}$) | | × |

Figure 4.4: Liabilities for claim $claim_{NoRoom}$

The claim can occur either if $WebComp$ or $IntSys$ does not forward the reservation request (first line); or if $CompHotel$ does not send the confirmation of the reservation to $WebComp$ (second line). Liabilities for these two errors are associated respectively with Thomas Cook and iBis.

Machine *Liabilities* (Figure 4.5) defines the liabilities (constant *Liability*) as a function that maps the declared claim to the errors (represented by parametric properties) and the corresponding liable parties.

MACHINE *Liabilities*
SEES $ComponentsAPI$, $LogFiles$, $LogProperties$, $ContractAgents$, $Claims$
CONSTANTS *Liability*
PROPERTIES

```
/* liabilities */
```

$Liability \in DECLARED\_CLAIMS \rightarrow (PAR\_PROP \nrightarrow \mathcal{F}(SIGN\_PARTY)) \wedge$

$(c1)$ $\forall(claim, error).(claim \in \mathsf{dom}(Liability) \wedge error \in \mathsf{dom}(Liability(claim)) \Rightarrow$
$PPropComps(ExpressClaim(claim)) \subseteq PPropComps(error)) \wedge$

$Liability = \{\dots\}$

END

Figure 4.5: Machine *Liabilities*

For a given claim *claim*, the domain of $Liability(claim)$ should be the set of possible errors associated to this claim. The components attached to error specifications must be included in the components attached to the property describing the claim ($c1$).

**Example 4.3.** Liabilities for $claim_{NoRoom}$

$$Liability(claim_{NoRoom}) = \{parProp_{NoForward} \mapsto \{ThomasCook\}, \\ parProp_{NoConfirm} \mapsto \{iBis\}\}$$

The above definition represents the table of Figure 4.4.

### 4.2.4 B Machine and Technical Annexes

Machines *ContractAgents* and *Claims* are part of Annex A of the contract and should be completed to specify respectively the agents and the claims for which the parties wish to specify liabilities. Machine *Liabilities* should be completed to specify the liabilities tables of each claim appearing in Annex E.

The following table summarizes the relevant information about the machines and technical annexes:

| Machine | Annex | Information described | To be instantiated? |
|---|---|---|---|
| *ContractAgents* | A | Agents of the contract | Yes |
| *Claims* | A | Claim definitions | Yes |
| *Liabilities* | E | Liabilities | Yes |

## 4.3 Establishing Liabilities

In the following sections, we specify the four steps of the log analysis procedure of Figure 4.1 (Sections 4.3.1 to 4.3.4) and the interpretation of the results (Section 4.3.5).

### 4.3.1 Step 1: Log Collection

**INPUT**: a claim instance $(claim, par)$ and a set of logs $(LogSet)$

Given a claim instance, the property attached to the claim is first instantiated using the sequence of parameters $par$. Let $claim = (plain, def, parProp)$, $prop_{claim}$ is them built as follows:

$$prop_{claim} = (PPropPredicate(parProp))(par)$$

The definition of $CLAIM\_INST$ ensures that the parameters of the claim instance $(par)$ can be used to instantiate $parProp$ (condition $c1$ in machine *Claims* in Section 4.2.2). Then, it is necessary to verify if the logs in the input $(LogSet)$ contain the entries related to the components of $prop_{claim}$.

**OUTPUT**: Yes if the following holds for $LogSet$:

$$PropComps(prop_{claim}) \subseteq \mathsf{union}(\{comps \mid \exists(cont).((comps, cont) \in LogSet)\})$$

No, otherwise

### 4.3.2 Step 2: Log Validity Analysis

The second step consists in analyzing *LogSet* to check that it meets the requirements on our models.

**INPUT**: *LogSet*, the input from the previous step.

For all $log \in LogSet$, check the proof obligation $\text{log} \in LOG\_FILE$. According to the $LOG\_FILE$ definition, for each $\text{log} = (comps, cont)$ the entries must be unique ($cont \in \mathsf{iseq}(ENTRY)$) and comply with the API definition for the component. In addition, all entries must be related to *comps* and respect the ordering between associated *Send* and *Rec* (conditions $c1$ and $c2$ in machine $LogFiles$ in Section 3.3.2).

**OUTPUT**: Yes/No

### 4.3.3 Step 3: Claim Validity Analysis

The third step consists in applying the log analyzer to *LogSet* to determine if the claim is confirmed by the logs.

**INPUT**: *LogSet*, from the input of the first step; and $prop_{claim}$ instantiated in the first step.

Compute the set of scenarios *scen* and *ok* as follows:

$$scen, ok \leftarrow VerifyProperty(LogSet, prop_{claim})$$

**OUTPUT**: *scen* the set of scenarios and *ok* the subset of scenarios in which the claim is observed.

The decision to accept or reject the claim is based on the sets *scen* and *ok*, as explained in Section 3.5.

### 4.3.4 Step 4: Liability Analysis

The fourth step consists in searching in the *ok* scenarios for the errors that are related to the liabilities. This step also includes the computation of the parties liable for each scenario.

INPUT: $(claim, par)$, the claim instance given as input of step 1.
*ok*, computed in the previous step.

The algorithm used to compute liabilities is the following one:

$scenAnalysis := \emptyset$;
for all $log \in ok$ and $error \in \mathsf{dom}(Liability(claim))$ do
  $prop_{error} := (PPropPredicate(error))(par)$;
  $scen_{error}, ok_{error} \leftarrow VerifyProperty(\{log\}, prop_{error})$;
  if $scen_{error} = ok_{error}$ then
    $scenAnalysis := scenAnalysis \cup \{(log \mapsto error)\}$;
  end if
end for

OUTPUT: $scenAnalysis \in LOG\_FILE \leftrightarrow PAR\_PROP$, the relationship between the scenarios and errors.

*scenAnalysis* relates the scenarios where the claim is accepted ($log \in ok$) with the errors defined in Annex E ($error \in \mathsf{dom}(Liability(claim))$). Because we apply $VerifyProperty$ to a set reduced to a singleton, we have $scen_{error} = \{log\}$. Then the condition $scen_{error} = ok_{error}$ expresses the fact that the error occurs in the scenario or not. In the last case *scenAnalysis* is equal to $\emptyset$.

### 4.3.5 Interpreting the Results

The result *scenAnalysis* can be presented in the form of a table as follows:

| Errors | Scenarios in $ok$ | | | | |
|---|---|---|---|---|---|
| | $log_1$ | $log_2$ | ... | $log_{n-1}$ | $log_n$ |
| $error_1$ | $LP_{1,1}$ | $\emptyset$ | ... | $\emptyset$ | $\emptyset$ |
| $error_2$ | $\emptyset$ | $LP_{2,2}$ | ... | $\emptyset$ | $LP_{2,n}$ |
| ... | $\emptyset$ | $\emptyset$ | ... | $\emptyset$ | $\emptyset$ |
| $error_m$ | $\emptyset$ | $LP_{m,2}$ | ... | $\emptyset$ | $\emptyset$ |

Figure 4.6: Table representing *scenAnalysis*

This table defines the analysis of a given *claim*. Each value $LP_{i,j}$ represents the set of liable parties for an occurrence of $error_i$ in $log_j$. We have for $log_j$:

$$error_i \in \mathsf{dom}(Liability(claim)) \Rightarrow LP_{i,j} = Liability(claim)(error_i)$$

If $error_i$ does not occur in $log_j$ then $LP_{i,j} = \emptyset$. For a given scenario $log_j$, the set of liable parties can be computed as follows:

$$liabilityLog_j := \bigcup_{i=1}^{m} LP_{i,j}$$

Deciding which parties should be liable for a claim is based on the table representing *scenAnalysis*. The most trivial case is when the same parties are liable for all scenarios. If this is not the case, then similarly to the analysis of the claim (step 3) there are two options. If only a subset of the logs has been collected then the analyst can return to the log collecting step to add new logs in order to try to reduce the number of scenarios. If all logs have been analyzed, then the contract stipulates that the parties should resort to an expert to establish liabilities based on the logs (or any other resources) and the tables in Annex E.

**Example 4.4.** Establishing liabilities

Suppose that, after applying the log analysis procedure for an instance of the claim $claim_{NoRoom}$, we obtain the following table:

| Errors | Scenarios | | | |
|---|---|---|---|---|
| | $scenario_1$ | $scenario_2$ | $scenario_3$ | $scenario_4$ |
| $parProp_{NoForward}$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $parProp_{NoConfirm}$ | $\{iBis\}$ | $\{iBis\}$ | $\{iBis\}$ | $\{iBis\}$ |
| *liabilityLog* | $\{iBis\}$ | $\{iBis\}$ | $\{iBis\}$ | $\{iBis\}$ |

In this case, it is clear that the component *CompHotel* has not sent a confirmation to the client ($parProp_{NoConfirm}$) in all possible scenarios and then agent *iBis* is liable.

## 4.4 Log Distribution Analysis

In this section, we specify a way to analyze log distributions in order to increase the strength of the logs used as digital evidence.

### 4.4.1 Technical and Legal Assumptions

The logging protocols described in Section 1.5 provide guarantees (integrity, confidentiality and authentication) that can increase the probative values of the logs used as digital evidence. These guarantees are usually based on security properties of cryptographic techniques and communication protocols. However, in a legal procedure, the probative value of digital evidence is always left to the appraisal of the judge (usually based on the advice of technical experts). In order to reduce legal uncertainties in this procedure, we propose a way to evaluate a given log distribution which is based on the assumption that agents only tamper with the logs when there is a possibility to change the result of the analysis of a claim (accept a claim that should be rejected or vice-versa) in their favor. We first introduce some technical concepts related to log distribution that complement the notions presented in Section 3.3.3.

The malicious attacks considered here are related to the type and action fields of entries. We do not consider attacks consisting in changing the value of the parameters. Therefore, we define the notion of "form" of a log entry as a pair made of its type and action. The function $EvForm$ maps every log entry to its form:

$$EvForm \in ENTRY \rightarrow (TYPE \times ACTION)$$
$$\forall (tp, c_S, c_R, ac, par).((tp, c_S, c_R, ac, par) \in ENTRY \Rightarrow$$
$$EvForm(tp, c_S, c_R, ac, par) = (tp, ac))$$

In order to define the level of trustworthiness of a set of log files, we specify the function $Access$ that maps every form of log entry into the set of agents that have access to this entry (which are also the agents in charge of logging the corresponding entries).

$$Access \in (TYPE \times ACTION) \rightarrow \mathcal{F}(AGENT)$$

For example, we specify that BNP has access to the logs of $CompBank$ as:

$$Access(Rec, Debit) = \{BNP\}$$
$$Access(Send, Bill) = \{BNP\}$$

We also assume that certain agents can be trusted not to tamper with certain types of entries. We represent this assumption by the function $Trust$ that maps every form of entry with agents that are trusted for this form.

$$Trust \in (TYPE \times ACTION) \rightarrow \mathcal{F}(AGENT)$$

Legally speaking, this assumption can be seen as a presumption of integrity of the log entries either due to technical reasons (e.g. use of trusted modules) or legal reasons, (e.g. strong regulations). For example, we assume that BNP can be trusted not to tamper with the entries of $CompBank$ because banks are subject to rigorous regulations and they are very unlikely to breach them.

Finally, some logging protocols described in Section 1.5 include the notion of authenticated log entries, e.g. [New & Rose 2001, Sackmann et al. 2006, Ma & Tsudik 2009]. We also define the function $Auth$ that maps each action to the set of agents that authenticate this action:

$$Auth \in ACTION \rightarrow \mathcal{F}(AGENT)$$

The meaning of an authenticated action is that it is "testified" by some agents and it cannot be tampered with by any other agents, e.g. the testifying agent digitally sign the entry with the his private key. As an illustration, suppose that the entry $NewRequest$ is authenticated by $Client$, then only $Client$ may forge a reservation request.

Note that the functions $Access$, $Trust$ and $Auth$ have to be specified for every action but it is possible to map actions to empty sets to indicate, respectively, actions that are not logged, actions that do not have any trusted agent or actions that are logged without any authentication.

### 4.4.2 Malicious Attacks

As mentioned in Section 1.5.2, integrity is a crucial condition to ensure the probative values of the logs. However, in order to reason about the integrity of the log files, we must first define the types of malicious attacks to be addressed.

We consider two types of attacks against the logs: the deletion of an entry and the addition of a new entry. The corresponding attacks are denoted respectively by:

- $Delete(agent, comp, i)$ – deletion by *agent* of the $i$-th entry in the log file of the component *comp*.
- $Add(agent, comp, i, en)$ – addition by *agent* of entry *en* at position $i$ in the log file of the component *comp*.

In [Le Métayer et al. 2010b], we define the effect of each of these attacks. Typically the goal of these attacks can be to escape potential claims from other agents or to build claims against other agents. We assume that an attack can happen only from agents that are not trusted for the targeted entries ($agent \notin Trust(EvForm(en))$) and that can benefit from this attack. The evaluation of this benefit is based on the potentiality to make it possible (or impossible) to sustain claims in favor (or against) *agent*.

More specifically, an attack *Delete* takes place only if there exists a set of log files and a claim instance such that one of the following conditions happens:

- *agent* is the **plaintiff** and the claim would be rejected with the original log files, but **would be accepted** after deletion of the entry.
- *agent* can be **liable** for the claim and the claim would be accepted for the original log files, but **would be rejected** after deletion of the entry.

*Add* attacks are defined similarly with the additional condition that only agents that authenticate a log entry can add it to the log file. This condition expresses the fact that agents cannot forge log entries authenticated by other agents.

### 4.4.3 Claim Events

To analyze the acceptability of claims we represent the claim instances by "claim events" taking the form $C(plain, def, prop)$, such that *plain* is the plaintiff, $def$ the defendant and *prop* is the ground for the claim. The set $CLAIM\_EV$ represents the set of all claim events considered by parties. We also use a function $Eval \in (PROP \times LOG\_SET) \rightarrow \{Ac, Rj, In\}$ that returns, for a given property and a set of logs, $Ac$ (accepted) if the property holds, $Rj$ (rejected) if it does not hold or $In$ if it is inconclusive. We add the possible result $In$ to represent the fact that it is not always possible to establish liabilities using the logs (e.g. logs may have been tampered with or the scenario analysis may not provide a conclusive result).

$Eval$ can be defined as follow based on $VerifyProperty$ introduced in Chapter 3:
$VerifyProperty(LogSet, prop) = (scen, ok)$ then $Eval(prop, LogSet) =$

- $Ac$ if $scen = ok$
- $Rj$ if $ok = \emptyset$
- $In$ otherwise

Note that we use the claim event $C(plain, def, prop \wedge error)$ to encapsulate the conditions for the validity of the claim ($prop$) and for the liabilities of an agent ($error$). For example, for a given $claim = (plain, def, prop)$ such that:

$$Liability(claim)(error) = \{Ag_1, Ag_2\}$$

we use claim events $C(plain, Ag_1, prop \wedge error)$ and $C(plain, Ag_2, prop \wedge error)$.

### 4.4.4 Acceptable Log Distribution

We define a notion of acceptable log distribution that depends on the technical and legal assumptions (Section 4.4.1) and the malicious attacks (Section 4.4.2). This definition provides the conditions on log distributions to ensure that the attacks of malicious agents on the logs do not have any impact on the validity of the claims. It is based on the "dependency function" $Neutral \in (TYPE \times ACTION) \rightarrow \mathcal{F}(AGENT)$. $Neutral(tp, ac)$ returns the set of agents which can be considered as neutral for a form of entry $(tp, ac)$ because the occurrence of these entries is neither detrimental nor beneficial for claims in which these agents might be involved:

**Definition 3.** $Neutral$
**agent** $\in Neutral(tp, ac)$ iff:

$\forall(LogSet, LogSet', en).(LogSet \in LOG\_SET \wedge LogSet' \in LOG\_SET \wedge$
$en \in ENTRY \wedge EvForm(en) = (tp, ac) \wedge$
$ExtractEn(LogSet, en) = ExtractEn(LogSet', en) \Rightarrow \forall(def, plain, prop).$
$(C(\mathbf{agent}, def, prop) \in CLAIM\_EV \Rightarrow Eval(prop, LogSet) = Eval(prop, LogSet') \wedge$
$C(plain, \mathbf{agent}, prop) \in CLAIM\_EV \Rightarrow Eval(prop, LogSet) = Eval(prop, LogSet')))$

Where function $ExtractEn(LogSet, en)$ returns the log files of $LogSet$ from which all occurrences of entries $en$ have been removed.

We also define the function $Fplus$, which is weaker than $Neutral$: $Fplus(tp, ac)$ returns the set of agents for which the occurrence of entries of the form $(tp, ac)$ cannot be *detrimental* in the sense that they cannot contribute to make a claim against them valid or a claim from them invalid:

**Definition 4.** *Fplus*
**agent** $\in Fplus(tp, ac)$ iff:

$\forall(LogSet, LogSet', en).(LogSet \in LOG\_SET \wedge LogSet' \in LOG\_SET \wedge$
$en \in ENTRY \wedge EvForm(en) = (tp, ac) \wedge ContainEv(LogSet, en) \wedge$
$ExtractEn(LogSet, en) = ExtractEn(LogSet', en) \Rightarrow \forall(def, plain, prop).$
$(C(\mathbf{agent}, def, prop) \in CLAIM\_EV \Rightarrow Eval(prop, LogSet') \Rightarrow Eval(prop, LogSet) \wedge$
$C(plain, \mathbf{agent}, prop) \in CLAIM\_EV \Rightarrow Eval(prop, LogSet) \Rightarrow Eval(prop, LogSet')))$

where function $ContainEv(LogSet, en)$ returns true if there is a log file in $LogSet$ that contains an occurrence of entry $en$.

The dual of $Fplus(tp, ac)$ is $Fminus(tp, ac)$ which returns the set of agents for which the occurrence of entries of the form $(tp, ac)$ cannot be *beneficial*.

**Definition 5.** *Fminus*
**agent** $\in Fminus(tp, ac)$ iff:

$\forall(LogSet, LogSet', en).(LogSet \in LOG\_SET \wedge LogSet' \in LOG\_SET \wedge$
$en \in ENTRY \wedge EvForm(en) = (tp, ac) \wedge \neg ContainEv(LogSet, en) \wedge$
$ExtractEn(LogSet, en) = ExtractEn(LogSet', en) \Rightarrow \forall(def, plain, prop).$
$(C(\mathbf{agent}, def, prop) \in CLAIM\_EV \Rightarrow Eval(prop, LogSet') \Rightarrow Eval(prop, LogSet) \wedge$
$C(plain, \mathbf{agent}, prop) \in CLAIM\_EV \Rightarrow Eval(prop, LogSet) \Rightarrow Eval(prop, LogSet')))$

Based on the definition of these three functions we define a log distribution as acceptable.

**Definition 6.** Acceptable log distribution

A log distribution is acceptable if and only if for any entry form $(tp, ac)$ such that $\exists(agent).(agent \notin Neutral(tp, ac))$ then $Access(tp, ac) \neq \emptyset$, and:

- $\forall(agent).(agent \in Access(tp, ac) \Rightarrow agent \in Neutral(tp, ac) \cup Trust(tp, ac))$

  or

- $Auth(ac) \neq \emptyset \wedge \forall(agent).(agent \in Auth(ac) \Rightarrow agent \in Fminus(tp, ac)) \wedge$
  $\forall(agent).(agent \in Access(tp, ac) \Rightarrow (agent \in Fplus(tp, ac) \vee agent \in Trust(tp, ac)))$

In other words, a log distribution is acceptable if each entry which may have an impact on a claim ($\exists(agent).(agent \in Neutral(tp, ac))$) is logged by at least one agent ($Access(tp, ac) \neq \emptyset$) and:

- the logging agents are neutral w.r.t this form of entry or can be trusted to log this form of entry

  or

- this form of entry is not beneficial for agents that authenticate them and not detrimental to logging agents, unless they can be trusted to log them.

Definition 6 can constitute the basis for an analyzer taking as input a log distribution and returning either a positive answer if the log distribution is acceptable or the set of entries for which the log distribution is not acceptable (with the set of agents concerned) otherwise. The only difficulty is the computation of $Neutral$, $Fplus$ and $Fminus$ which depend on the function $Eval$. This step can be more or less challenging (or require more or less approximations) depending on the expressive power of the language of properties. For a language involving only existential properties on entries (occurrence or absence of entries) this analysis can be done easily based on the association of a polarity with each individual entry depending on its context of occurrence in a claim property (which can be positive, negative or neutral).

### 4.4.5 Results

The ultimate goal of the log distribution is to ensure that any claim can be evaluated correctly. Correctness means that any claim evaluation is correct, even when the logs have been subject to malicious attacks. We assume the existence of a function $\Phi$ that receives a set of logs and a sequence of attacks (of type $Add$ or $Delete$) and returns the set of logs after the attacks.

**Definition 7.** Correct log distribution
A log distribution is correct iff:

$\forall(LogSet, LogSet', Attacks, claimEv, plain, def, prop).$
$(claimEv \in CLAIM\_EV \wedge claimEv = C(plain, def, prop) \wedge LogSet' =$
$\Phi(LogSet, Attacks) \Rightarrow$
$Eval(prop, LogSet) = Eval(prop, LogSet')$

The intuition of the above definition is that in a correct log distribution every claim event is evaluated to the same result before and after the logs have been tampered. The main property of acceptable log distributions is stated as follow:

**Property 4.1.** Any acceptable log distribution is correct.

This property follows from correctness and consistency properties established in similar settings in [Le Métayer et al. 2010b] (with a slightly different notation). The consistency property ensures that no attack against the logs can introduce inconsistencies between the logs of different agents.

### 4.4.6 Case Study

Let us consider an example of log distribution analysis using our case study. We assume that, although $WebComp$ has been provided by the travel agency, it is the client has access to the logs of $WebComp$. The travel agency ($ThomasCook$) has access to the logs of its internal system ($IntSys$), the hotel ($iBis$) to the logs of the reservation component ($CompHotel$) and the bank ($BNP$) to the logs of the payment component:

$Dist = \{\{WebComp\}, \{IntSys\}, \{CompHotel\}, \{CompBank\}\}$
$Access = \{$
$(Rec, NewRequest) \mapsto \{Client\}$, $(Rec, CancelRequest) \mapsto \{Client\}$,
$(Send, Request) \mapsto \{Client\}$, $(Send, Cancel) \mapsto \{Client\}$,
$(Send, Book) \mapsto \{ThomasCook\}$, $(Send, Unbook) \mapsto \{ThomasCook\}$,
$(Rec, Request) \mapsto \{ThomasCook\}$, $(Rec, Cancel) \mapsto \{ThomasCook\}$,
$(Rec, Book) \mapsto \{iBis\}$, $(Rec, Unbook) \mapsto \{iBis\}$,
$(Send, Response) \mapsto \{iBis\}$, $(Rec, Response) \mapsto \{ThomasCook\}$,
$(Send, Confirm) \mapsto \{iBis\}$, $(Send, Debit) \mapsto \{iBis\}$,
$(Rec, Debit) \mapsto \{BNP\}$, $(Send, Bill) \mapsto \{BNP\}$
$\}$

The types of entries associated with the client himself[1] $(Send, NewRequest)$ and $(Rec, Bill)$ are not logged, therefore they do not appear in the definition of $Access$.

We consider in a first stage that none of the entries are authenticated, i.e.:

$$\forall(ac).(ac \in ACTION \Rightarrow Auth(ac) = \emptyset)$$

Finally, we assume that the bank and the hotel are trusted to log their debit because they have to follow very stringent regulatory requirements. This assumption can be expressed as follows:

$$Trust(Send, Debit) = \{iBis\}$$
$$Trust(Rec, Debit) = \{BNP\}$$

Let us consider the claim $C(Client, ThomasCook, prop)$ raised by a client $clientId$with the property $prop$ stating that the client has been charged for a reservation that has not been requested. More precisely, the predicate of this property is defined as follows:

$PropPredicate(prop) =$
$\lambda(log).(log \in LOG\_FILE \wedge \exists(cont).(log = (\{IntSys, CompHotel\}, cont)) \mid$
$\mathsf{bool}(\exists(cont).(log = (\{IntSys, CompHotel\}, cont) \wedge$
$(Rec, WebComp, IntSys, Request, [sessionId, clientId, details]) \notin \mathsf{ran}(cont) \wedge$
$(Send, CompHotel, CompBank, Debit, [sessionId, hotelId, clientId, price]) \in \mathsf{ran}(cont)))$

[1] As opposed to the $WebComp$ component which logs its entries

The predicate holds if the logs contain a *Debit* but no corresponding *Request*.

The application of the criteria of Definition 6 shows that the log distribution is not acceptable because the entries of form $(Rec, Request)$ (which are involved in the predicate of *prop*) are not appropriately logged:

- The agent *ThomasCook* that has access to this type of entry is neither trusted nor neutral because it belongs neither to $Trust(Rec, Request)$ nor to $Neutral(Rec, Request)$.
- This type of entry is not authenticated by any agent, because $Auth(Request) = \emptyset$.

One possible option to enhance the log distribution is that *Request* entries are authenticated by *Client*. Then, we have:

$$Auth(Request) = \{Client\}$$

and the second criterion of Definition 6 is satisfied. More precisely, we have:

$Auth(Request) \neq \emptyset \wedge$
$\forall(agent).(agent \in Auth(Request) \Rightarrow agent \in Fminus(Rec, Request)) \wedge$
$\forall(agent).(agent \in Access(Rec, Request) \Rightarrow \ agent \ \in \ Fplus(Rec, Request))$

*ThomasCook* has still access to the entries of the form $(Rec, Request)$ but cannot add any entries because they must be authenticated by *Client*.

This example shows how the definition of acceptable log distribution can be used to provide logs that are more likely to be accepted as digital evidence because it is possible to show that no agent can modify the logs to get any benefit in the treatment of the claims. Of course, the logs must in addition be protected against external attacks using secure logging methods (Section 1.5).

### 4.4.7 Related Works

The work which is the closest in spirit to the approach presented in this section is the SLAng formalist (Section 1.4.8). The notion of monitorability in the contract formalist of SLAng differs from our notion of acceptable logs in the sense that monitorability concerns the possibility for an agent to get trustable information about the execution of the system rather than the availability of evidence. In other words, monitorability ensures that the agent can trust the information, but not that he can use this information to convince a third party (e.g. a judge). Another, more general, departure from [Skene et al. 2007] has to do with the objective itself: the goal of [Skene et al. 2007] is to analyze contracts to check their monitorability by the parties whereas we take contracts as granted and analyze the log distribution to check that it is sufficient to sustain the potential claims between

the parties. The two approaches are clearly complementary and could be integrated in a common environment.

Theories of accountability and audit have been proposed in the context of security and usage policies ([Jagadeesan et al. 2009, Vaughan et al. 2008, Cederquist et al. 2007]). The logging mechanisms and audit procedures are defined in a logical framework (based on a policy language) and applied to practical examples such as the protection of confidential documents. The main departure with respect to these approaches is the fact that we do not focus on security and usage policies and consider more generally claims between parties in a contract.

## 4.5 Contributions of the Chapter

Existing contract formalisms (Section 1.4) focus on the specification of the obligations of each party. Our first contribution is a generic way to specify liabilities for failures of the system using parametric properties (Section 4.2).

Another contribution is a procedure to establish liabilities in case of failure (Section 4.3). In [Hvitved 2010], the author points out that most contract formalisms do not provide any support to establish liabilities in case of breach of contract. In comparison with CSL (Section 1.4.4), instead of associating a party with each clause of the contract, we provide a way to specify liabilities which is related to the specification of the components (errors and failures) which allows us to define more complex liabilities where, for example, more than one party may be liable for a single failure.

In [Goessler et al. 2012] we discuss the causal relationship between claims and errors and we provide an alternative way to specify liabilities. The advantage of the approach presented in [Goessler et al. 2012] is that the definition of causality can be applied systematically to claims that are not predefined in the contract. However, this approach requires more effort in terms of specification because the analysis of causality is based on the specification of the correct and incorrect behaviors of the components. In practice, both methods are complementary: our approach can be followed to treat claims based on a priori allocation of liabilities and the approach in [Goessler et al. 2012] when the parties cannot foresee the potential errors which can lead to a failure.

Another contribution of our work is to take into account distributed logs and their analysis. We can establish liabilities even when the logs are recorded by different components (i.e. a central logging system is not mandatory) or when parts of the log files are not available. When the results of the log analysis are not conclusive, it may still be possible to use them to establish liabilities based on other factors (such as the likelihood of the scenarios).

The last contribution is the study of properties of a given log distribution in terms of acceptability (Section 4.4). This definition allows us to check if a distribution can provide trustworthy digital evidence, even when agents cannot be trusted to store the information in their logs or not to tamper with the log entries.

# Chapter 5

# Implementation of the Log Analysis Procedure

In this section we describe the implementation of the LAPRO (Log Analysis PROcedure) Tool that implements the Log Analysis Procedure.

Section 5.1 describes the language of properties used to define claims and liabilities in LAPRO. Starting from B models, Section 5.2 explains how sets and constants are represented. In Section 5.3 we describe the log analyzer algorithm dedicated to our language of properties and in Section 5.4 we present the LAPRO tool. Finally, in Section 5.5 we provide an evaluation of the performance of LAPRO for large logs.

## 5.1 Language of Properties

In our framework, properties define functions that map log files to boolean values. Up to this point, we have expressed these functions using first order logic. However, first order logic is undecidable and not specifically designed to reason about sequences. Its use as a language of properties also introduces complexity issues [Gupta 1992, Claessen et al. 2002]. The language of properties proposed in the literature to address these issues (Section 1.6) generally include limited versions of the operators of first order logic and introduce operators that provide a more natural way to express conditions about sequences (such as the operators of LTL, Section 1.6.3).

Regular expressions provide a way to define patterns in sequences of characters in UNIX systems [IEEE & The Open Group 1997]. According to [Vardi 2008], regular expressions have the advantage to be easily understood by computer science engineers and are thus well suited to industrial applications. Indeed, regular expressions are used in industrial specification languages and supported by programming languages widely used in industry, such as Python and Java.

Languages of properties usually combine boolean connectives ($\wedge, \vee, \neg$) and regular expressions [Vardi 2008, Leucker & Sánchez 2010]. This is the case, for example, in TBL (Section 1.6.5), PSL (Section 1.6.8) and LogScope (Section 1.6.7).

We define a language for expressing parametric properties on logs inspired by to the language LogScope (Section 1.6.7) that combines regular expressions and boolean operators. Properties are defined over entries which are expressed using the following grammar:

**Definition 8.** Grammar of entries

$$
\begin{aligned}
\text{<}entry\text{>} &::= \text{`(' <}type\text{> `,' <}text\text{> `,' <}text\text{> `,' <}text\text{> `,[' <}parList\text{> `])' | `e'} \\
\text{<}type\text{>} &::= \text{`Send' | `Rec'} \\
\text{<}text\text{>} &::= \texttt{[a-zA-Z0-1\_]+} \\
\text{<}parList\text{>} &::= \text{<}value\text{> | <}value\text{> `,'<}parList\text{>} \\
\text{<}value\text{>} &::= \text{<}text\text{> | `\{' <}text\text{> `\}' | `\$'}
\end{aligned}
$$

An entry (<*entry*>) is represented either by its values separated by commas (surrounded by parentheses `( )`) or the terminal `e` that represents any possible entry. A parameter that needs to be instantiated is represented by its identifier between curly brackets `{ }`. The terminal `$` represents any parameter value. Based on the above definition, we define regular expressions over entries as follows:

**Definition 9.** Regular expression

$$\text{<}RE\text{>} ::= \quad \text{<}entry\text{> | `('<}RE\text{>`)' | <}RE\text{><}RE\text{> | <}RE\text{>`|'<}RE\text{> | <}RE\text{>`*'}$$

Regular expressions (<*RE*>) can be grouped (using parentheses) and concatenated. The operator '`|`' separates alternatives. Finally, the operator '`*`' is the Kleene star that denotes zero or more occurrences of the regular expression. Classically, a given regular expression $RE$ defines a language $\mathcal{L}(RE)$. The only particularities of our language are the terminals `e` and `$`. We define theirs associated sets as follows:

$$
\begin{aligned}
\mathcal{L}(\texttt{e}) = \{&(tp, t_1, t_2, t_3, [par]) \mid tp \in \mathcal{L}(type) \wedge t_1 \in \mathcal{L}(text) \wedge t_2 \in \mathcal{L}(text) \wedge t_3 \in \mathcal{L}(text) \wedge \\
&par \in \mathcal{L}(parList)\} \\
\mathcal{L}(\texttt{\$}) = \{&t \mid t \in \mathcal{L}(text)\}
\end{aligned}
$$

We say that a log satisfies a regular expression $RE$ if it belongs to the language of the regular expression:

$$log \text{ satisfies } RE \Leftrightarrow log \in \mathcal{L}(RE)$$

As mentioned before, our language of properties combines regular expressions with boolean operators. A property can be expressed using the following grammar:

**Definition 10.** Language of properties

$$
\begin{aligned}
<prop> ::= \quad & <RE> \mid \\
& \text{`NOT(' } <prop> \text{ `)'} \mid \\
& \text{`('} <prop> \text{ `AND' } <prop> \text{ `)'} \mid \\
& \text{`(' } <prop> \text{ `OR' } <prop> \text{ `)'}
\end{aligned}
$$

The function `holds` takes a property written in the above language and a log and returns the boolean value "`true`" if the property holds for the log and "`false`" otherwise. We define this function recursively as follows:

**Definition 11.** Verification algorithm

$\texttt{holds}(prop, log) =$
  `if` $prop = RE$ `then`
    `return` $log \in \mathcal{L}(RE)$;
  `elsif` $prop = \texttt{NOT}(P)$ `then`
    `return` $\neg\texttt{holds}(P, log)$;
  `elsif` $prop = (P_1$ `AND` $P_2)$ `then`
    `return` $\texttt{holds}(P_1, log) \wedge \texttt{holds}(P_2, log)$;
  `else` $prop = (P_1$ `OR` $P_2)$ `then`
    `return` $\texttt{holds}(P_1, log) \vee \texttt{holds}(P_2, log)$;
  `end if`

**Example 5.1.** Regular expression

We use our language of properties to represent the predicate of the parametric property $parProp_{NoRoom}$ (Example 4.2 in Section 4.2.2) stating that the client has sent a request and has not received any confirmation (`clientId` and `sessionId` being two parameters):

```
(e*(Rec,Client,WebComp,NewRequest,[{sessionId},{clientId},$])e* AND
   NOT(e*(Send,CompHotel,CompBank,Confirm,[{sessionId},$,$])e*))
```

For any regular expression, it is possible to construct a non-deterministic finite automaton that accepts the same language. This automaton can be built in $O(n)$ time, where $n$ is the size of the regular expression and it processes each character of a string in $O(1)$ time. Thus, checking if a regular expression matches a string of size $m$ takes $O(n + m)$ time. Additionally, the automaton requires $O(n^2)$ memory [Sidhu & Prasanna 2001]. It has also been shown that regular expressions have greater expressiveness than LTL [Wolper 1983]. For example, given a string `S` the regular expression `(a.)*` ensures that `a` occurs every two character of the string, which cannot be expressed by an LTL formula.

## 5.2 Representation of Logs and Liabilities

In this section we provide more details on the implementation of LAPRO. We use the Python language to represent data and algorithms and HTML for the interface with the Apache web server. In this section we describe how sets and constants of B models are represented. Table 5.1 summarizes our implementation choices.

| Machine | Set/Constant | Implementation |
|---|---|---|
| *ComponentsAPI* | *COMP* | `set(string)` |
| | *ACTION* | `set(string)` |
| | *TYPE* | `set(string)` |
| | *PARAM* | `string` |
| *ContractAgents* | *AGENT* | `set(string)` |
| *ComponentsAPI* | *Interface* | `set(string,string)` |
| | *Invoke* | `set(string,string)` |
| | *NumParams* | `set(string,int)` |
| *LogDistribution* | *Dist* | `set(set(string))` |
| *ContractAgents* | *SIGN_PARTY* | `set(string)` |
| | *THIRD_PARTY* | `set(string)` |
| *LogFiles* | *ENTRY* | class `LogEntry` |
| | *LOG_FILE* | class `LogFile` |
| *LogOperations* | *LOG_SET* | class `LogSet` |
| *LogProperties* | *PROP* | class `LogProperty` |
| *ParametricProperties* | *PAR_PROP* | class `LogParProperty` |
| *Claims* | *CLAIM* | class `Claim` |
| | *CLAIM_INST* | class `ClaimInstance` |
| *Liabilities* | *Liability* | class `Liability` |

Table 5.1: Implementation of B machines

The enumerated sets (*COMP*, *ACTION*, *TYPE* and *AGENT*) are represented by sets of strings. Elements of the given set *PARAM* are strings of characters. Constants which are defined by the parties (such as *Interface*) are also represented by sets. The constants that define types (such as *ENTRY*) are represented by classes (one for each constant), each of them containing a method `verify`. The class diagrams of Figure 5.1 and Figure 5.2 show respectively the relationship between the classes pursuant to logs and liabilities.

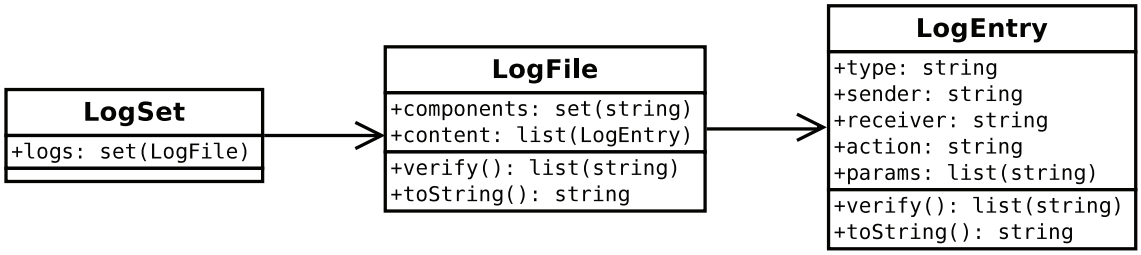



Figure 5.1: Classes defining logs

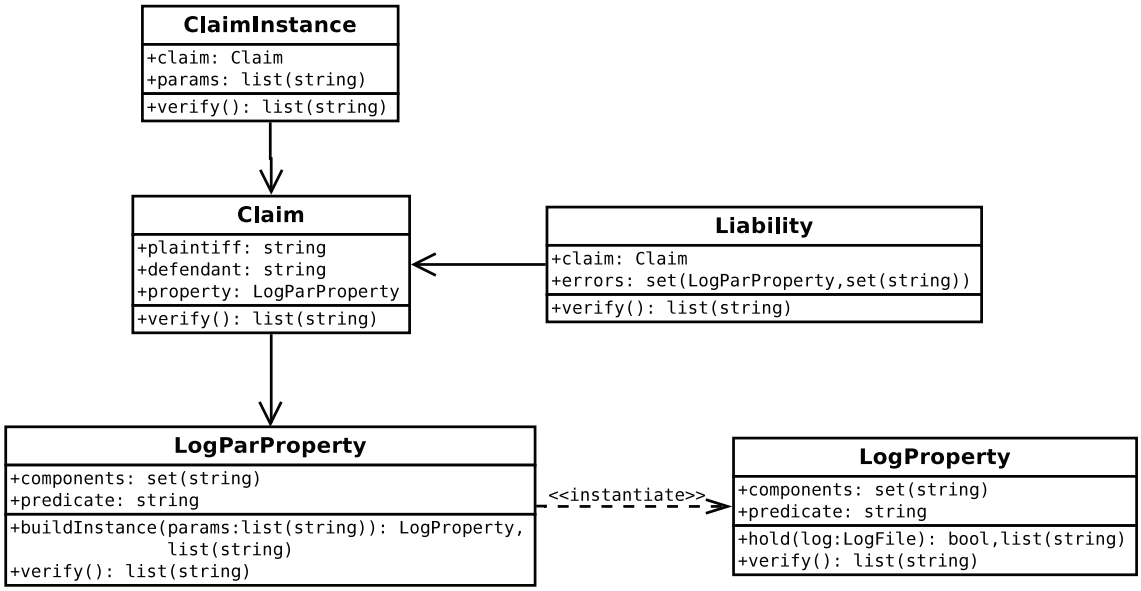



Figure 5.2: Classes defining liabilities

Each class contains a method `verify` that tests the validity of elements of classes, according to the B models. Conditions are summarized in Table 5.2.

| Class | Properties verified |
|---|---|
| `LogEntry` | - actions are in the interface of receiver<br>- actions may be invoked by sender<br>- the number of parameters is the same as specified in the API |
| `LogFile` | - entries are unique<br>- entries are related to the components attached to the log file<br>- *Send* entries occur before *Rec* entries<br>- conditions in the `verify` method of each entry in the content are satisfied |
| `LogSet` | - conditions in the `verify` method of each log file in the set are satisfied |
| `LogProperty` | - the syntax of the predicate is correct |
| `LogParProperty` | - the syntax of the predicate is correct |
| `Claim` | - the defendant is a signing party<br>- conditions in the `verify` method of the parametric property of claim are satisfied |
| `ClaimInstance` | - the number of parameters is equal to the number of parameters of the property attached to the claim |
| `Liability` | - liable parties are signing parties<br>- components associated with errors are included in the components of the properties attached to the claims<br>- conditions in the `verify` method of each parametric property expressing an error are satisfied |

Table 5.2: Conditions checked by `verify`

### 5.2.1 Declaration of Liabilities

Claims and liabilities can be declared using formatted text files. Functions `readClaim` and `readLiabilities` respectively build an instance of `Claim` and an instance of `Liability` from these files.

The format of claim declaration files is the following:

```
<comment describing the claim (optional)>
<plaintiff>
<defendant>
<components attached to the claim property, separated by comma>
<predicate of the property>
```

The first line may contain a comment (starting with #) with a description of the claim. The following four lines specify respectively the plaintiff, the defendant, the components and the predicate of the property. The predicate of the property is specified using the language presented in Section 5.1.

The format of the liability declaration file is the following:

```
<name of the claim file>
# error 1 description (optional)
<components attached to the error property, separated by comma>
<predicate of the property>
<liable parties associated with the error, separated by comma>
# error 2 description (optional)
...
```

The first line is the name of the claim declaration file for which liabilities are specified. Liabilities are defined by a sequence of error definitions. The first line of an error definition contains an optional comment (starting with #). The following three lines specify the error property (components and predicate) and the parties liable for the errors.

LAPRO offers an interface to be used during the elaboration of the contract to validate the files defining claims and liabilities (Figure 5.3). This interface allows us to load and check declaration files, according to the method `verify` (table 5.2).

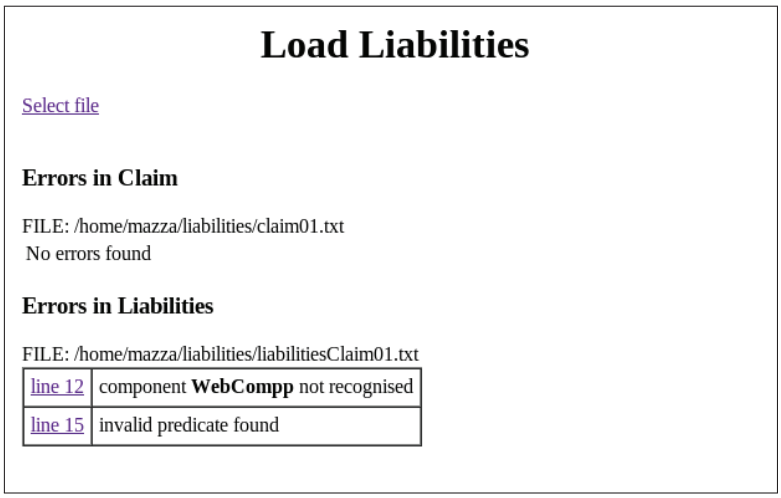



Figure 5.3: Load Liabilities Interface

There are two types of errors:

- *Name error* occurs if the name of a component, action or agent is not recognized.
- *Syntax error* occurs if the input file is not conform to the syntax described in Section 5.2.2. This is the case, for example, if the first line of a log file does not contain a list of components names.

The first column also contains a link to a document that displays the line which has led to the error.

### 5.2.2 Declaration of Log Files

Log files can also be declared using formatted text files. The first line contains the name of the components (separated by commas) related to the log. The following lines should contain the entries of the logs, one line per entry. Log entries are encoded according to the grammar given Section 5.1, excluding `e` and `$`. For example:

$$\begin{aligned} log = (&\{WebComp, IntSys\},\\ &[(Send, WebComp, IntSys, Request, [s011, emazza, 14Jun\_Paris]),\\ &(Rec, WebComp, IntSys, Request, [s011, emazza, 14Jun\_Paris]),\\ &(Send, IntSys, CompHotel, Book, [s011, emazza, 14Jun\_Paris])]) \end{aligned}$$

is encoded by:

```
WebComp,IntSys
Send,WebComp,IntSys,Request,[011,emazza,14Jun_Paris]
Rec,WebComp,IntSys,Request,[011,emazza,14Jun_Paris]
Send,IntSys,CompHotel,Book,[011,emazza,14Jun_Paris]
```

The function `readLogFile` takes as input a name of a log declaration file and builds an instance of class `LogFile`. LAPRO offers an interface to load log files (Section 5.4.1).

## 5.3 Log Analyzer Algorithm

In order to implement the log analyzer, we must first provide an implementation of the functions *Extract* and *Merge* (machine *LogOperations* in Section 3.4).

The function `extract` takes as input a `LogFile` and a set of component names and returns a new `LogFile` containing only the entries related to the given set of components. This function also returns an error message when the set of components is not included in the components of the log (according to the domain definition of the extract function in Section 3.4.1).

The function `merge` takes as input a `LogSet` and returns a new instance of `LogSet` containing all the scenarios produced. A naïve implementation of this function consists first in producing all possible interleavings of entries that respect the local order and them verifying which interleavings respect the causal order between *Send* and *Rec* entries. The problem with this approach is that the amount of interleavings grows exponentially with the number of entries.

Many solutions have been proposed to improve the efficiency of this process [Brightwell & Winkler 1991, Kalvin & Varol 1983, Varol & Rotem 1981]. We have implemented the algorithm proposed by [Varol & Rotem 1981] due to its simplicity and efficiency in practice [Pruesse & Ruskey 1994]. This algorithm consists in keeping track of every possible position that each entry can take relatively to other ones. Then, the algorithm uses these locations to swap entries and to produce the scenarios. For example, let $en_1$, $en_2$ and $en_3$ be three entries such that $en_1$ happens before $en_2$ and $en_3$. The algorithm first records that $en_1$ cannot swap with any other entry and $en_2$ can swap positions with $en_3$ and then it produces the scenarios $en_1, en_2, en_3$ and $en_1, en_3, en_2$. In terms of complexity, this algorithm takes at most $O(n)$ time per scenario where $n$ is the number of entries.

The log analyzer is implemented by the function `verifyProperty` that takes as input a `LogSet` and a `LogProperty` and returns two new instances of `LogSet` corresponding to the sets *scen* and *ok*. This function returns an error if the selected logs do not contain the entries associated with the set of components attached to the property (pre-condition of $VerifyProperty$ in Section 3.5.3).

The verification of a log property for a given log is made by parsing the predicate of the property according to the definition of `holds` (Definition 11). To verify that a given log file is

recognized by a regular expression, we use the regular expression library `re` of Python. This library includes a function `match` that, given a string and a regular expression, returns the position where the regular expression matches the string or `None` if the regular expression does not match the string. To build a string from an entry representation (Definition 8) we have to expand the non conventional operators of our regular expressions (the `$` and `e` symbols). Each symbol `$` is replaced by the string `\w*` with `\w` defined as `[a-zA-Z0-9_]`). Each `e` is replaced by the string `\((Send|Rec),\w+,\w+,\w+,\[[\w,]*\]\)`[1] that matches any entry.

## 5.4 Log Analysis PROcedure (LAPRO) Tool

In this section, we describe the LAPRO tool that implements the log analysis procedure, as specified Figure 4.1. The interface of LAPRO consists of several HTML documents, each of them corresponding to one step of the procedure. The links *Next Step* and *Previous Step* allow us to browse through the different steps of the procedure.

### 5.4.1 Step 1: Log Collection

The interface described in Figure 5.4 allows us to select the set of logs to be analyzed and the claim instance for which liabilities must be established. The link *Add log file* allows us to select the log files that should be in the format specified in Section 5.2.2. The link *Select liabilities* can be used to define liabilities in the format specified in Section 5.2.1. Then, the interface makes it possible to define the values of the parameters specific to the claim instance.

[1] Since the characters square brackets and parentheses have a special meaning in regular expressions if we wish to match then in the text we should note them with a backslash


Figure 5.4: Log Collection Interface

When the user tries to proceed to the next step, the procedure shows a warning (Figure 5.5) if the collected log files do not fulfill the conditions of table 5.2. LAPRO also produces an error message if a parameter is not defined.

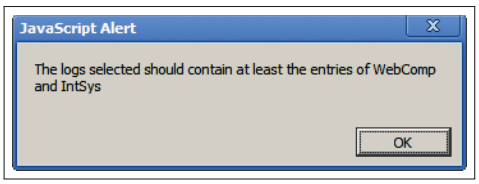



Figure 5.5: Alert Message in Log Collection

### 5.4.2 Step 2: Log Validity Analysis

In the second step (Figure 5.6), LAPRO verifies the log file format according to the syntax defined in Section 5.2 and conditions of Table 5.2. If any error occurs, the first column of each table contains the location of the error in the file and the second column contains the error message.

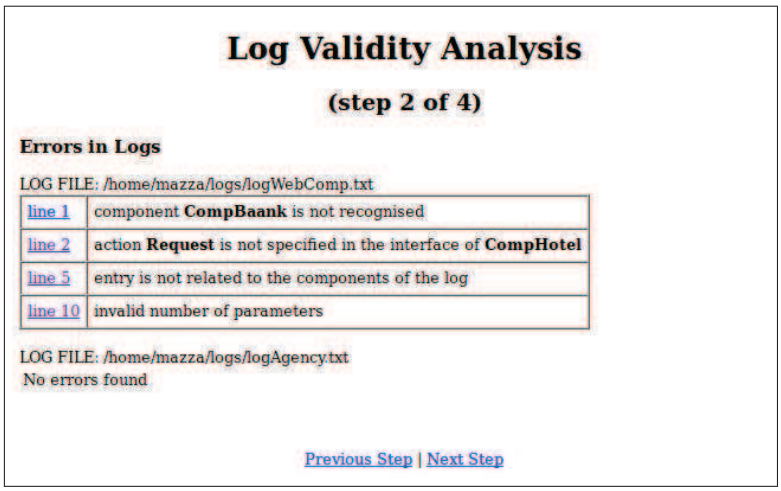



Figure 5.6: Log Validity Analysis Interface

Besides *name error* and *syntax error* (Section 5.2.1), *integrity errors* can occur if the properties defined in Table 5.2 do not hold. The first column contains a link to a document displaying the error, as shown Figure 5.7.


Figure 5.7: View Error Interface for Log Entry

### 5.4.3 Step 3: Claim Validity Analysis

In the third step, described Figure 5.8, LAPRO executes the merge and extraction to produce the scenarios and it applies the `verifyProperty` algorithm (Section 5.3) to determine the scenarios where the claim occurs. The document of Figure 5.8 contains a table listing the set of all scenarios (set *scen*) resulting from the log analyzer. Each scenario has one unique identifier (first column) with a link to a document displaying its content (Figure 5.9).

The second column indicates with a '×' the subset of scenarios where the claim is valid (set *ok*). The third column contains a link, *Exclude*, that allows us to exclude a given scenario (for instance a non realistic one).

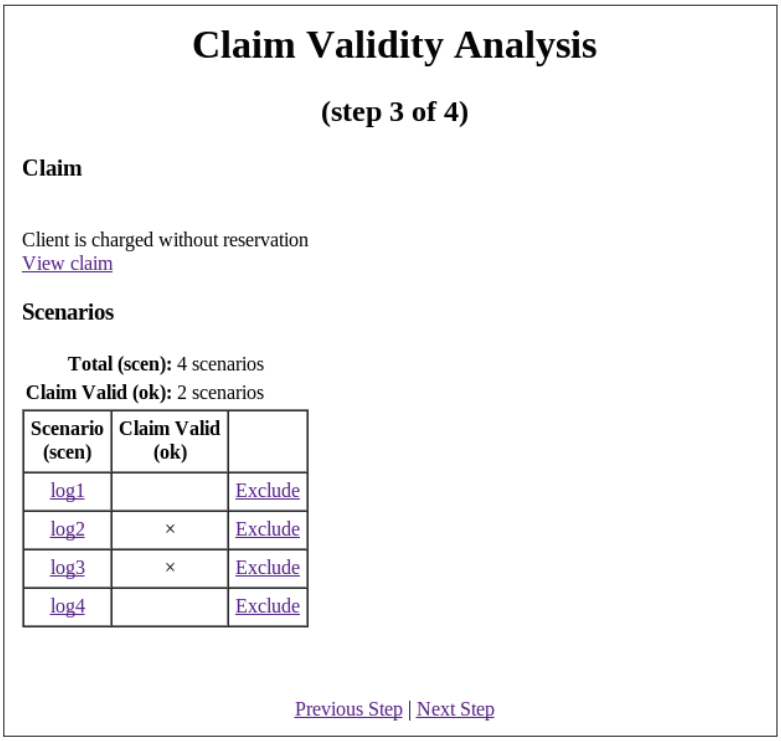



Figure 5.8: Claim Validity Analysis Interface

The link *View Claim* displays the components and predicate of the property attached to the analyzed claim.

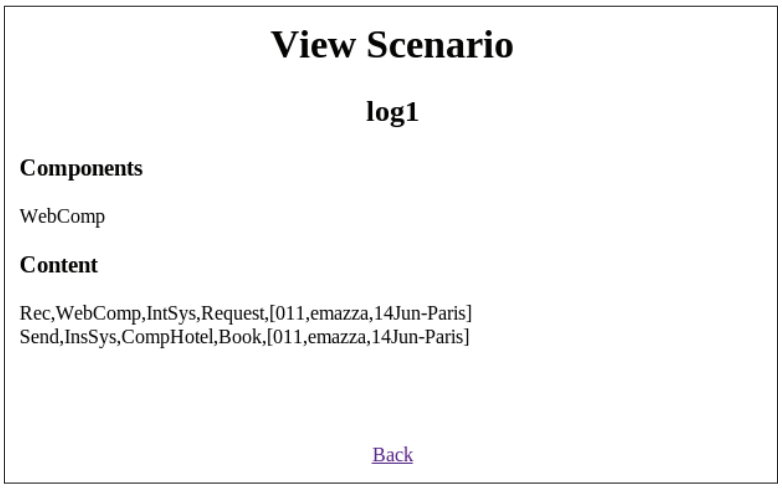



Figure 5.9: View Scenario Interface

### 5.4.4 Step 4: Liability Analysis

In the last step (Figure 5.10), LAPRO analyzes the marked scenarios to search the errors associated with the claim.

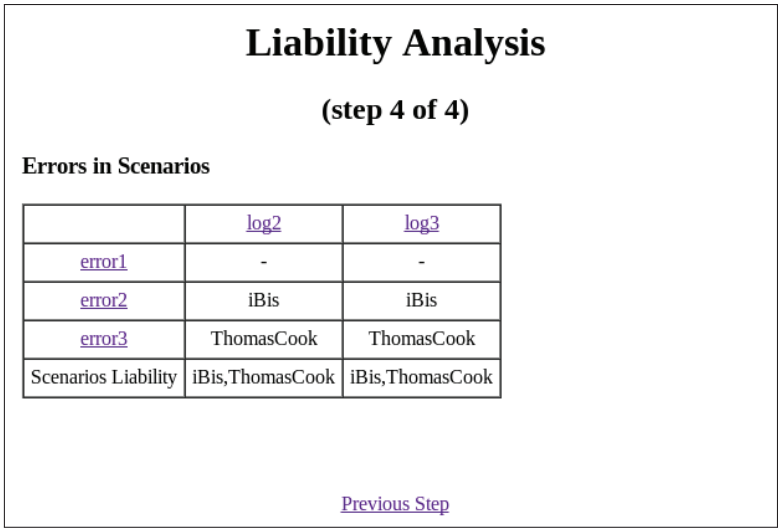



Figure 5.10: Liability Analysis Interface

The first column lists the set of errors coming from the input file describing liabilities. Each of the following columns represents one scenario ($log_i$) where the claim is valid ($log_i \in ok$). Each cell indicates the liable parties ('-' indicates that the error did not occur in

the scenario). The last line of the table contains the union of all parties liable for a given scenario.

Additionally, each error contains a link ($error_i$) to a document that displays a description, components and predicate attached to the error property (Figure 5.11).

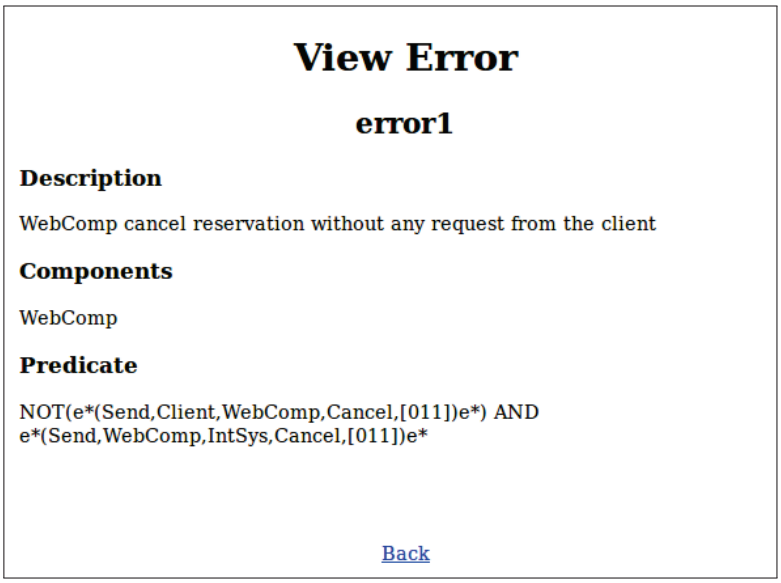



Figure 5.11: View Error Property Interface

## 5.5 LAPRO Evaluation

In this document, for the sake of conciseness, we only have presented the verification of the properties for small logs. However, in practice the number of events in the logs may be very large. In this section we evaluate the performance of our tool for the analysis of large logs.

The three most relevant criteria for this evaluation are the following:

1. First, the computational cost to evaluate if a given log is valid (step 2). More precisely, we must verify if the log content fulfills the properties defined in the set $LOG\_FILE$ (method `verify` of the class `LogFile`). The more complex condition is the verification that *Send* entries always occur before their respective *Rec* entries. The other conditions consists in verifying each entry of the log content to check if the action and number of parameters are specified as defined in the API of the components.

2. Second, the cost of verifying a property for a large log file. We want to verify the time necessary to verify properties written in the language of Section 5.1.

3. Third, the cost of merging the logs and producing the set of scenarios. When working with large amounts of entries, the number of scenarios may also be very large. We must determine if the number of scenario is too large to be computed in practice.

The evaluations reported in the next sections were performed in a machine with a Intel Core i5 1.07Ghz and 4GB of memory running Ubuntu 11.10.

### 5.5.1 Evaluation of `LogFile.verify()`

We randomly produced logs containing errors that should be detected by the method `verify`. The evaluation was performed on a set of log files containing a total of $10^3$, $10^4$ and $10^5$ entries. For each of these sizes, 100 log files were produced.

The following table shows the average time necessary to execute the method `verify` (Table 5.2) on the class `LogFile`:

| `verify` average time (in milliseconds) | | |
|---|---|---|
| $10^3$ entries | $10^4$ entries | $10^5$ entries |
| 11 | 117 | 1240 |

All conditions of the method `verify` take linear time to be performed because each entry only needs to be verified once. To verify the order between *Send* and *Rec* entries the tool stores each entry and its position in a hashtable structure and checks for every *Rec* entry if its *Send* counterpart has a lower position in the log file.

### 5.5.2 Evaluation of log merging

Based on our case study, we randomly produced logs with the entries of multiple sessions that could occur in parallel and we measured the time necessary to produce all scenario. We considered the log distributions that could produce the largest numbers of scenario for the components $WebComp$, $IntSys$ and $CompHotel$:

$$Dist = \{\{WebComp\}, \{IntSys\}, \{CompHotel\}\}$$

The following table shows the results of the scenario analysis:

| number of scenarios (total time to compute scenarios in seconds) | | |
|---|---|---|
| 15 entries | 20 entries | 25 entries |
| $1.55 \times 10^5 scenarios (< 1)$ | $5.25 \times 10^7 (87)$ | *timeout* |

Additionally, the average time necessary to produce each scenario is always less than 1 millisecond, even for a set of log files with a total of $10^5$ entries. However, one problem that we faced was the number of scenarios produced that grows exponentially with the number of entries (for logs with 25 entries it was not possible to compute the number of scenario event after several minutes). Optimizations to solve this problem are discussed in Section 5.5.4.

### 5.5.3 Evaluation of the verification of properties

We randomly produced logs containing a total of $10^3$, $10^4$ and $10^5$ entries. For each of these sizes, 100 log files were produced; then we measured the average time necessary to evaluate the properties $parProp_{NoRoom}$ (Section 5.1) and $prop_{LateCancel}$ (Section 3.8). These properties were chosen because they include all the operators of the language proposed in Section 5.1 (boolean operators, regular expressions using Kleene star and entries ordering). The log files were produced in a way such that the entries of the section for which the failure occurred would be randomly added to the log file and the properties tested would hold.

The predicate $parProp_{NoRoom}$ (Example 5.1) is represented as follows:

```
(e*(Rec,Client,WebComp,NewRequest,[{sessionId},{clientId},$])e* AND
   NOT(e*(Send,CompHotel,CompBank,Confirm,[{sessionId},$,$])e*))
```

and $prop_{LateCancel}$ is expressed as follows:

```
          e*(Rec,Client,WebComp,CancelRequest,[{sessionId}])e*
(Send,CompHotel,CompBank,Debit,[{sessionId},{hotelId},{clientId},$])e*
```

The following table shows the average time necessary to analyze $parProp_{NoRoom}$ and $prop_{LateCancel}$.

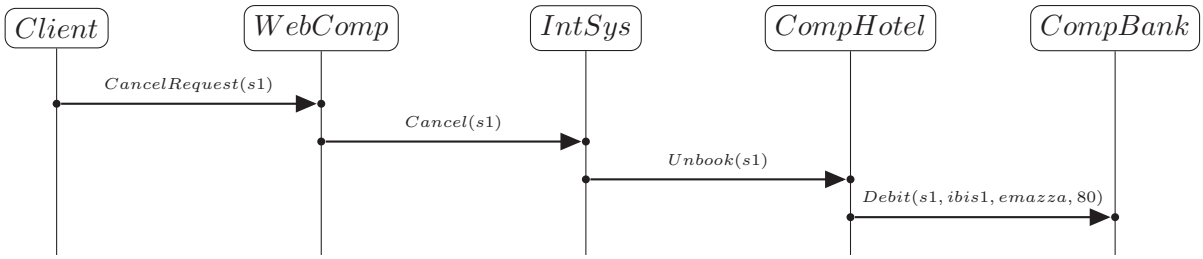
| Property | Average time (in miliseconds) | | |
|---|---|---|---|
| | $10^3$ entries | $10^4$ entries | $10^5$ entries |
| $parProp_{NoRoom}$ | 6 | 58 | 594 |
| $prop_{LateCancel}$ | 4 | 40 | 457 |

### 5.5.4 Optimizations

Regarding the results of the evaluation of LAPRO, we may conclude that only efficiency problem comes from the production of the scenarios. Some optimizations may be implemented to address this issue. The first one consists in filtering the log content to keep only the entries that are relevant for the property. Filters may be based on specific parameters. For example, we may filter only entries relative to a single session or perform the analysis only with entries that occur between the first (*NewRequest*) and the final entries (*Debit*) of a given session.

Another possible optimization is to check, before producing the scenarios, if the property does not state conditions about the order of the entries (as an illustration, the property $parProp_{NoRoom}$ only states conditions on the existence of entries). If this is the case, then the verification of the property may be performed using a single scenario because the properties of the function *Merge* ensures that the scenarios contain the same entries, but in different orders.

Finally, in some cases we may conclude that the order of two given entries must be the same for scenarios by analyzing the causality between *Send* and *Rec* entries (based on the relation "happened before" [Lamport 1978]). As an illustration, let us consider the logs containing the entries pictured in the following diagram:




Even if other entries occurred before or after the above entries, the property $prop_{LateCancel}$ (given the session $s1$ and client $emazza$) holds for every scenario because the causality of the *Send*/*Rec* entries shows that the *CancelRequest* occurred before *Debit*.

An issue about the last two optimizations is that it may be difficult to automatically detect for a given property if it states conditions about the ordering of the entries because these conditions are written within regular expressions that may also express conditions about the existence of entries and parameter values. This issue can be addressed by adding in the language an operator stating the ordering between two entries and using this operator, instead of regular expressions, to express temporal conditions.

# Conclusion

The core of this thesis is a formal framework to help the parties to specify software liabilities in precise way and to establish these liabilities in case of failure. This framework is based on a model of software contracts (Chapter 2) that was developed in the context of the LISE project (Section 1.7). Our framework addresses the three main challenges identified in Section 1.3:

1. **How to represent liabilities in a precise and unambiguous way?**

   Chapter 4 presents the formal models that allow us to associate failures of the system with the parties that should be considered liable for these failures (Section 5.2.1). This approach takes into account the legal contractual requirements, as analyzed by the lawyers of the LISE project.

   To validate our approach we propose a property language (Section 5.1) and a file format to specify liabilities (Section 5.2.1) that can be annexed to the contract.

2. **How to produce digital evidence to establish liabilities?**

   Chapter 4 defines desirable properties of log architectures (Section 4.4) that are likely to increase the acceptability of the logs as digital evidence [Le Métayer et al. 2010b]. The definition of acceptable log architectures is based on the interest of the parties to tamper with the logs in order to change the evaluation of future claims in their favor.

3. **How to establish liabilities in case of incident?**

   Chapter 4 defines the log analysis procedure to establish liabilities. This procedure is based on a log analyzer (Section 3.5.3) and it can be applied to distributed logs [Mazza et al. 2010]. The analysis can be performed using only a subset of the logs and we also propose an incremental version of the log analyzer [Mazza et al. 2010].

   To validate our approach, Chapter 5 provides an implementation of the log analyzer and the log analysis procedure (LAPRO tool in Section 5.4). The implementation also includes the models necessary to represent the logs and their content (Section 5.2.2). Experimental results of its application to different types of logs are also presented and its performances are analyzed (Section 5.5).

## Perspectives

The work presented in this thesis can be pursued in several directions. First, the models of log properties presented in Chapter 3 could be generalized to consider logs that may not be complete with respect to a given property because the property depends on future events. One possibility consists in using a 3-value logic (true/false/unknown) with 'unknown' representing a property which cannot be evaluated because it depends on future events. Another alternative is to define a function returning the part of the logs (e.g. initial and final entries) necessary to evaluate a property.

A second line of work, which follows from Chapter 4, is the analysis of the causality between the failures and the errors in the logs. This kind of analysis would provide a way for the parties to define their respective liabilities in a more direct and logical way. In practice, it would also avoid the need to define once for all in the contract the association of errors to liabilities. An initial discussion of this subject is presented in [Goessler et al. 2012].

Another avenue for further research is the refinement of the definition of acceptable log architectures to consider levels of acceptability. We may define the acceptability of the claims differently for each party based on the degree of trust between agents.

With respect to Chapter 5, the property specification language could be extended to include conditions about parameters, which would make it possible to filter the relevant parts of the logs. We also wish to extend LAPRO to suggest possible acceptable log distributions when the initial log distribution is not acceptable.

Finally, although our work has been made in collaboration with lawyers and presented to legal experts[2], we think that our approach may benefit form a formal validation by professionals lawyers in charge of writing contracts.

[2]Several workshop and conferences were organized by the LISE project involving many legal experts and lawyers. The program of these conferences can been found at `http://licit.inrialpes.fr/lise/`

# Bibliography

[ABC 2010] ABC (2010). Dow Suddenly Drops 1,000 Points on Worries Over Greek Debt, Then Recovers (available at: abcnews.go.com/Business/story?id=10576136). [Online].

[Abrial 1996] Abrial, J. (1996). *The B-Book*. Cambridge University Press.

[Accorsi 2006] Accorsi, R. (2006). On the relationship of privacy and secure remote logging in dynamic systems. In S. Fischer-Hübner, K. Rannenberg, L. Yngström, & S. Lindskog (Eds.), *SEC*, volume 201 of *IFIP* (pp. 329–339). Springer.

[Accorsi 2009] Accorsi, R. (2009). Safe-keeping digital evidence with secure logging protocols: State of the art and challenges. In O. Goebel, R. Ehlert, S. Frings, D. Günther, H. Morgenstern, & D. Schadt (Eds.), *IMF* (pp. 94–110). IEEE Computer Society.

[Andersen et al. 2006] Andersen, J., Elsborg, E., Henglein, F., Simonsen, J. G., & Stefansen, C. (2006). Compositional specification of commercial contracts. *STTT*, 8(6), 485–516.

[Anderson & Moore 2009] Anderson, R. & Moore, T. (2009). Information Security: Where Computer Science, Economics and Psychology Meet. *Philosophical Transactions of the Royal Society A: Mathematical Physical & Engineering Sciences*, 367(1898), 2717–2727.

[Anderson 2008] Anderson, R. J. (2008). Information security economics - and beyond. In R. van der Meyden & J. van der Torre (Eds.), *DEON*, volume 5076 of *Lecture Notes in Computer Science* (pp.49). Springer.

[Arasteh et al. 2007] Arasteh, A. R., Debbabi, M., Sakha, A., & Saleh, M. (2007). Analyzing Multiple Logs for Forensic Evidence. *Digital Investigation*, 4, 82–91.

[Armoni et al. 2002] Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M. Y., & Zbar, Y. (2002). The forspec temporal logic: A new temporal property-specification language. In J.-P. Katoen & P. Stevens (Eds.), *TACAS*, volume 2280 of *Lecture Notes in Computer Science* (pp. 296–211). Springer.

[Avizienis et al. 2004] Avizienis, A., Laprie, J.-C., Randell, B., & Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1), 11–33.

[Badeau & Amelot 2005] Badeau, F. & Amelot, A. (2005). Using b as a high level programming language in an industrial project: Roissy val. In H. Treharne, S. King, M. C. Henson, & S. A. Schneider (Eds.), *ZB*, volume 3455 of *Lecture Notes in Computer Science* (pp. 334–354). Springer.

[BalaBit IT Security 2011] BalaBit IT Security (2011). *Syslog-ng web site*. Available at: www.balabit.com/network-scerity/syslog-ng.

[Barringer et al. 2010a] Barringer, H., Groce, A., Havelund, K., & Smith, M. H. (2010a). Formal Analysis of Log Files. *Aerospace Computing, Information, and Communication*, 7.

[Barringer et al. 2007] Barringer, H., Rydeheard, D., & Havelund, K. (2007). Rule Systems for Run-Time Monitoring: from Eagle to Ruler. In *Proceedings of the 7th international conference on Runtime verification*, RV'07 (pp. 111–125).

[Barringer et al. 2010b] Barringer, H., Rydeheard, D. E., & Havelund, K. (2010b). Rule systems for run-time monitoring: from eagle to ruler. *J. Log. Comput.*, 20(3), 675–706.

[Bauer et al. 2011] Bauer, A., Leucker, M., & Schallhart, C. (2011). Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4), 14.

[Bauer et al. 2006] Bauer, A., Leucker, M., & Streit, J. (2006). Salt - structured assertion language for temporal logic. In Z. Liu & J. He (Eds.), *ICFEM*, volume 4260 of *Lecture Notes in Computer Science* (pp. 757–775). Springer.

[Behm et al. 1999] Behm, P., Benoit, P., Faivre, A., & Meynadier, J.-M. (1999). Météor: A successful application of b in a large project. In J. M. Wing, J. Woodcock, & J. Davies (Eds.), *World Congress on Formal Methods*, volume 1708 of *Lecture Notes in Computer Science* (pp. 369–387). Springer.

[Berry 2007] Berry, D. M. (2007). Abstract appliances and software: The importance of the buyer's warranty and the developer's liability in promoting the use of systematic quality assurance and formal methods. CiteSeerX - Scientific Literature Digital Library and Search Engine, http://www.scientificcommons.org/42749418.

[Birsch 2004] Birsch, D. (2004). Moral responsibility for harm caused by computer system failures. *Ethics and Information Technology*, 6, 233–245. 10.1007/s10676-005-5609-5.

[Bitan 2004] Bitan, H. (2004). Les clauses limitatives des responsabilité dans les contrats informatique. *Communication Commerce Eletronique*, 1, 14–19.

[Blom et al. 2004] Blom, J., Hessel, A., Jonsson, B., & Pettersson, P. (2004). Specifying and generating test cases using observer automata. In J. Grabowski & B. Nielsen (Eds.), *FATES*, volume 3395 of *Lecture Notes in Computer Science* (pp. 125–139). Springer.

[Bollig & Leucker 2003] Bollig, B. & Leucker, M. (2003). Deciding ltl over mazurkiewicz traces. *Data Knowl. Eng.*, 44(2), 219–238.

[Branch 2010] Branch, R. A. I. (2010). *Derailment of a Docklands Light Railway Train Near West India Quay Station*. Technical report, Department for Transport, UK.

[Brightwell & Winkler 1991] Brightwell, G. & Winkler, P. (1991). Counting linear extensions. *Order*, 8, 225–242. 10.1007/BF00383444.

[Buskirk & Liu 2006] Buskirk, E. V. & Liu, V. T. (2006). Digital evidence: Challenging the presumption of reliability. *J. Digital Forensic Practice*, 1(1), 19–26.

[Cardoso & Oliveira 2009] Cardoso, H. L. & Oliveira, E. C. (2009). Monitoring cooperative business contracts in an institutional environment. In J. Cordeiro & J. Filipe (Eds.), *ICEIS (2)* (pp. 206–211).

[Carrier 2003] Carrier, B. D. (2003). Defining digital forensic examination and analysis tool using abstraction layers. *International Journal of Digital Evidence*, 1(4).

[Cederquist et al. 2007] Cederquist, J. G., Corin, R., Dekker, M. A. C., Etalle, S., den Hartog, J. I., & Lenzini, G. (2007). Audit-based compliance control. *Int. J. Inf. Sec.*, 6(2-3), 133–151.

[Chang & Ren 2007] Chang, F. & Ren, J. (2007). Validating system properties exhibited in execution traces. In R. E. K. Stirewalt, A. Egyed, & B. Fischer (Eds.), *ASE* (pp. 517–520). ACM.

[Charette 2005] Charette, R. N. (2005). Why software fails. *IEEE Spectrum*.

[Chen & Roşu 2009] Chen, F. & Roşu, G. (2009). Parametric Trace Slicing and Monitoring. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (ETAPS 2009),* (pp. 246–261). Springer-Verlag.

[Chin et al. 2009] Chin, K.-S., Wang, Y.-M., Poon, G. K. K., & Yang, J.-B. (2009). Failure mode and effects analysis using a group-based evidential reasoning approach. *Computers & OR*, 36(6), 1768–1779.

[Claessen et al. 2002] Claessen, K., Hahnle, R., Martensson, J., & Ab, S. (2002). *Verification of Hardware Systems with First-Order Logic*. Technical report, Copenhagen, DIKU, University of Copenhagen, Denmark.

[ClearSy 2011] ClearSy (2011). Atelier b, version 4.0 (www.atelierb.eu).

[Coulouris et al. 2011] Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed systems - concepts and designs (5. ed.)*. International computer science series. Addison-Wesley-Longman.

[Dadeau & Tissot 2009] Dadeau, F. & Tissot, R. (2009). jsynopsys - a scenario-based testing tool based on the symbolic animation of b machines. *Electr. Notes Theor. Comput. Sci.*, 253(2), 117–132.

[Daskalopulu 2001] Daskalopulu, A. (2001). Modelling legal contracts as processes. *CoRR*, cs.AI/0106010.

[Desai et al. 2008] Desai, N., Narendra, N. C., & Singh, M. P. (2008). Checking correctness of business contracts via commitments. In L. Padgham, D. C. Parkes, J. Müller, & S. Parsons (Eds.), *AAMAS (2)* (pp. 787–794). IFAAMAS.

[DeYoung et al. 2010] DeYoung, H., Garg, D., Jia, L., Kaynar, D. K., & Datta, A. (2010). Experiences in the logical specification of the hipaa and glba privacy laws. In E. Al-Shaer & K. B. Frikken (Eds.), *WPES* (pp. 73–82). ACM.

[D'Souza 2003] D'Souza, D. (2003). A logical characterisation of event clock automata. *Int. J. Found. Comput. Sci.*, 14(4), 625–640.

[Falcone et al. 2009] Falcone, Y., Fernandez, J.-C., & Mounier, L. (2009). Runtime verification of safety-progress properties. In S. Bensalem & D. Peled (Eds.), *RV*, volume 5779 of *Lecture Notes in Computer Science* (pp. 40–59). Springer.

[Farrell et al. 2005] Farrell, A. D. H., Sergot, M. J., Sallé, M., & Bartolini, C. (2005). Using the event calculus for tracking the normative state of contracts. *Int. J. Cooperative Inf. Syst.*, 14(2-3), 99–129.

[Fenech et al. 2009a] Fenech, S., Pace, G. J., & Schneider, G. (2009a). Automatic conflict detection on contracts. In M. Leucker & C. Morgan (Eds.), *ICTAC*, volume 5684 of *Lecture Notes in Computer Science* (pp. 200–214). Springer.

[Fenech et al. 2009b] Fenech, S., Pace, G. J., & Schneider, G. (2009b). CLAN: A Tool for Contract Analysis and Conflict Discovery. In Z. Liu & A. P. Ravn (Eds.), *ATVA*, volume 5799 of *Lecture Notes in Computer Science* (pp. 90–96). Springer.

[Garg et al. 2011] Garg, D., Jia, L., & Datta, A. (2011). Policy auditing over incomplete logs: theory, implementation and applications. In Y. Chen, G. Danezis, & V. Shmatikov (Eds.), *ACM Conference on Computer and Communications Security* (pp. 151–162). ACM.

[Genicon 2008] Genicon, T. (2008). Le régime des caluses limitatives de réparation: état des lieux et perspectives. *Revue des contrats*, 3, 982–1008.

[Gerhards 2001] Gerhards, R. (2001). *RFC 3164: The BSD Syslog Protocol*. Available at: tools.ietf.org/html/rfc3164.

[Giannikis & Daskalopulu 2011] Giannikis, G. K. & Daskalopulu, A. (2011). Normative conflicts in electronic contracts. *Electronic Commerce Research and Applications*, 10(2), 247–267.

[Goessler et al. 2012] Goessler, G., Le Métayer, D., Mazza, E., Potet, M.-L., & Astefanoaei, L. (2012). Apport des méthodes formelles dans l'exploitation de logs informatiques dans un contexte contractuel. In *Approches Formelles dans l'Assitance au Développement de Logiciels (AFADL)*.

[Governatori 2005] Governatori, G. (2005). Representing business contracts in RuleML. *Int. J. Cooperative Inf. Syst.*, 14(2-3), 181–216.

[Governatori et al. 2008] Governatori, G., Hoffmann, J., Sadiq, S. W., & Weber, I. (2008). Detecting regulatory compliance for business process models through semantic annotations. In D. Ardagna, M. Mecella, & J. Yang (Eds.), *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing* (pp. 5–17). Springer.

[Governatori & Milosevic 2006] Governatori, G. & Milosevic, Z. (2006). A formal analysis of a business contract language. *Int. J. Cooperative Inf. Syst.*, 15(4), 659–685.

[Governatori & Pham 2009] Governatori, G. & Pham, D. H. (2009). Dr-contract: An architecture for e-contracts in defeasible logic. *International Journal of Business Process Integration and Management*, 5(4).

[Gupta 1992] Gupta, A. (1992). Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2/3), 151–238.

[Hallal et al. 2006] Hallal, H., Boroday, S., Petrenko, A., & Ulrich, A. (2006). A Formal Approach to Property Testing in Causally Consistent Distributed Traces. *Formal Aspects of Computing*, 18(1), 63–83.

[Hallal et al. 2003] Hallal, H., Boroday, S., Ulrich, A., & Petrenko, A. (2003). An automata-based approach to property testing in event traces. In D. Hogrefe & A. Wiles (Eds.), *TestCom*, volume 2644 of *Lecture Notes in Computer Science* (pp. 180–196). Springer.

[Henglein et al. 2009] Henglein, F., Larsen, K. F., Simonsen, J. G., & Stefansen, C. (2009). Poets: Process-oriented event-driven transaction systems. *J. Log. Algebr. Program.*, 78(5), 381–401.

[Holt 2006] Holt, J. E. (2006). Logcrypt: forward security and public verification for secure audit logs. In R. Buyya, T. Ma, R. Safavi-Naini, C. Steketee, & W. Susilo (Eds.), *ACSW Frontiers*, volume 54 of *CRPIT* (pp. 203–211). Australian Computer Society.

[Hvitved 2010] Hvitved, T. (2010). A trace-based model for multi-party contracts. In *Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS)*.

[IEEE & The Open Group 1997] IEEE & The Open Group (1997). *The Single Unix Specification, Version 2*. IEEE.

[Insa 2006] Insa, F. (2006). The admissibility of electronic evidence in court (a.e.e.c.): Fighting against high-tech crime - results of a european study. *J. Digital Forensic Practice*, 1(4), 285–289.

[IST 2011] IST (2011). IST Contract Porject (ist-contract.org).

[Ivanov 2005] Ivanov, L. (2005). Modeling and verification of a distributed transmission protocol. In L. T. Yang, H. R. Arabnia, Y. Li, S. N. Salloum, & J. G. Delgado-Frias (Eds.), *CDES* (pp. 64–70). CSREA Press.

[Jaffuel & Legeard 2007] Jaffuel, E. & Legeard, B. (2007). Leirios test generator: Automated test generation from b models. In J. Julliand & O. Kouchnarenko (Eds.), *B*, volume 4355 of *Lecture Notes in Computer Science* (pp. 277–280). Springer.

[Jagadeesan et al. 2009] Jagadeesan, R., Jeffrey, A., Pitcher, C., & Riely, J. (2009). Towards a theory of accountability and audit. In M. Backes & P. Ning (Eds.), *ESORICS*, volume 5789 of *Lecture Notes in Computer Science* (pp. 152–167). Springer.

[jDREW 2011] jDREW (2011). A Java Deductive Reasoning Engine for the Web (jdrew.org).

[Jones et al. 2003] Jones, S. P., Eber, J.-M., & Seward, J. (2003). How to write a financial contract. In Gibbons & de Moor (Eds.), *The Fun of Programming*. Palgrave Macmillan.

[Kalvin & Varol 1983] Kalvin, A. D. & Varol, Y. L. (1983). On the generation of all topological sortings. *J. Algorithms*, 4(2), 150–162.

[Keller & Ludwig 2003] Keller, A. & Ludwig, H. (2003). The wsla framework: Specifying and monitoring service level agreements for web services. *J. Network Syst. Manage.*, 11(1), 57–81.

[Kelsey et al. 2009] Kelsey, J., Callas, J., & Clemm, A. (2009). *Signed syslog messages*. Available at: tools.ietf.org//html/draft-ietf-syslog-sign-29.txt.

[Kenneally 2004] Kenneally, E. (2004). Digital logs - proof matters. *Digital Investigation*, 1(2), 94–101.

[Kent & Souppaya 2006] Kent, K. & Souppaya, M. (2006). *Guide to Computer Security Log Management*. Technical report, National Institute of Standards and Technology.

[Kyas et al. 2008] Kyas, M., Prisacariu, C., & Schneider, G. (2008). Run-time monitoring of electronic contracts. In S. D. Cha, J.-Y. Choi, M. Kim, I. Lee, & M. Viswanathan (Eds.), *ATVA*, volume 5311 of *Lecture Notes in Computer Science* (pp. 397–407). Springer.

[Lamanna et al. 2003] Lamanna, D. D., Skene, J., & Emmerich, W. (2003). Slang: A language for defining service level agreements. In *FTDCS* IEEE Computer Society.

[Lamport 1978] Lamport, L. (1978). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), 558–565.

[Landwehr 2009] Landwehr, C. E. (2009). A national goal for cyberspace: Create an open, accountable internet. *IEEE Security & Privacy*, 7(3), 3–4.

[Le Métayer et al. 2010a] Le Métayer, D., Maarek, M., Mazza, E., Potet, M.-L., Viet Triem Tong, V., Craipeau, N., Frénot, S., & Hardouin, R. (2010a). Liability in Software Engineering: Overview of the LISE Approach and Illustration on a Case Study. In *International Conference on Software Engineering (ICSE)* (pp. 135–144).

[Le Métayer et al. 2010b] Le Métayer, D., Mazza, E., & Potet, M.-L. (2010b). Designing log architectures for legal evidence. In J. L. Fiadeiro, S. Gnesi, & A. Maggiolo-Schettini (Eds.), *SEFM* (pp. 156–165). IEEE Computer Society.

[Leucker & Sánchez 2010] Leucker, M. & Sánchez, C. (2010). Regular linear-time temporal logic. In N. Markey & J. Wijsen (Eds.), *TIME* (pp. 3–5). IEEE Computer Society.

[Leucker & Schallhart 2009] Leucker, M. & Schallhart, C. (2009). A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5), 293–303.

[Leuschel & Butler 2003] Leuschel, M. & Butler, M. (2003). ProB: A model checker for B. In K. Araki, S. Gnesi, & D. Mandrioli (Eds.), *FME 2003: Formal Methods*, LNCS 2805 (pp. 855–874). Springer-Verlag.

[Ma & Tsudik 2009] Ma, D. & Tsudik, G. (2009). A new approach to secure logging. *TOS*, 5(1).

[Marotta-Wurgler 2007] Marotta-Wurgler, F. (2007). What's in a standard form contract? an empirical analysis of software license agreements. *Journal of Empirical Legal Studies*, 4(4), 677–713.

[Maurer 2004] Maurer, U. M. (2004). New approaches to digital evidence. *Proceedings of the IEEE*, 92(6), 933–947.

[Mazza et al. 2010] Mazza, E., Potet, M.-L., & Le Métayer, D. (2010). A formal framework for specifying and analyzing logs as electronic evidence. In J. Davies, L. Silva, & A. da Silva Simão (Eds.), *SBMF*, volume 6527 of *Lecture Notes in Computer Science* (pp. 194–209). Springer.

[Meyer et al. 1994] Meyer, J. J. C., Dignum, F. P. M., & Wiering, R. J. (1994). *The Paradoxes of Deontic Logic Revisited: A Compute Science Perspective.*

[Mittal & Garg 2001] Mittal, N. & Garg, V. K. (2001). Computation slicing: Techniques and theory. In J. L. Welch (Ed.), *DISC*, volume 2180 of *Lecture Notes in Computer Science* (pp. 78–92). Springer.

[Molina-Jiménez et al. 2009] Molina-Jiménez, C., Shrivastava, S. K., & Strano, M. (2009). Exception handling in electronic contracting. In B. Hofreiter & H. Werthner (Eds.), *CEC* (pp. 65–73). IEEE Computer Society.

[Morin-Allory et al. 2007] Morin-Allory, K., Fesquet, L., Roustan, B., & Borrione, D. (2007). Asynchronous online-monitoring of logical and temporal assertions. In *FDL* (pp. 286–290). ECSI.

[New & Rose 2001] New, D. & Rose, M. (2001). *Reliable Delivery for syslog.* Available at: tools.ietf.org/rfc/rfc3195.txt.

[Ohtaki 2008] Ohtaki, Y. (2008). Partial disclosure of searchable encrypted data with support for boolean queries. In *ARES* (pp. 1083–1090). IEEE Computer Society.

[Oren et al. 2008] Oren, N., Panagiotidi, S., Vázquez-Salceda, J., Modgil, S., Luck, M., & Miles, S. (2008). Towards a formalisation of electronic contracting environments. In J. F. Hübner, E. T. Matson, O. Boissier, & V. Dignum (Eds.), *COIN AAMAS*, volume 5428 of *Lecture Notes in Computer Science* (pp. 156–171). Springer.

[Pace & Schneider 2009] Pace, G. J. & Schneider, G. (2009). Challenges in the specification of full contracts. In M. Leuschel & H. Wehrheim (Eds.), *IFM*, volume 5423 of *Lecture Notes in Computer Science* (pp. 292–306). Springer.

[Paschke 2005] Paschke, A. (2005). Rbsla a declarative rule-based service level agreement language based on ruleml. In *CIMCA/IAWTIC* (pp. 308–314). IEEE Computer Society.

[Paschke & Bichler 2005] Paschke, A. & Bichler, M. (2005). Sla representation, management and enforcement. In *EEE* (pp. 158–163). IEEE Computer Society.

[Patel 2006] Patel, V. (2006). *The Contract Management Benchmark Report: Procurement Contracts*. Technical report, Aberdeen Group.

[Patel 2007] Patel, V. (2007). *Contract Lifecycle Management and the CFO: Optimizing Revenues and Capturing Savings*. Technical report, Aberdeen Group.

[Peláez & Bowles 1996] Peláez, C. E. & Bowles, J. B. (1996). Using fuzzy cognitive maps as a system model for failure modes and effects analysis. *Inf. Sci.*, 88(1-4), 177–199.

[Pnueli 1977] Pnueli, A. (1977). The temporal logic of programs. In *FOCS* (pp. 46–57). IEEE.

[Prakken & Sergot 1996] Prakken, H. & Sergot, M. J. (1996). Contrary-to-duty obligations. *Studia Logica*, 57(1), 91–115.

[Prisacariu & Schneider 2007] Prisacariu, C. & Schneider, G. (2007). A formal language for electronic contracts. In M. M. Bonsangue & E. B. Johnsen (Eds.), *FMOODS*, volume 4468 of *Lecture Notes in Computer Science* (pp. 174–189). Springer.

[Pruesse & Ruskey 1994] Pruesse, G. & Ruskey, F. (1994). Generating linear extensions fast. *SIAM J. Comput.*, 23(2), 373–386.

[Raskin & Schobbens 1999] Raskin, J.-F. & Schobbens, P.-Y. (1999). The logic of event clocks - decidability, complexity and expressiveness. *Journal of Automata, Languages and Combinatorics*, 4(3), 247–286.

[Rehm 2010] Rehm, J. (2010). Proved development of the real-time properties of the ieee 1394 root contention protocol with the event-b method. *Software Tools for Technology Transfer (STTT)*, 12(1), 39–51.

[Reith et al. 2002] Reith, M., Carr, C., & Gunsch, G. H. (2002). An examination of digital forensic models. *IJDE*, 1(3).

[Richard III & Roussev 2006] Richard III, G. G. & Roussev, V. (2006). Next-generation digital forensics. *Commun. ACM*, 49(2), 76–80.

[Rodin 2011] Rodin (2011). RODIN - Rigorous Open Development Environment for Complex Systems (rodin.cs.ncl.ac.uk).

[Rose 2001] Rose, M. (2001). *The Blocks Extensible Exchange Protocol Core*. Available at: tools.ietf.org/rfc/rfc3080.txt.

[RuleML 2011] RuleML (2011). The Rule Markup Initiative (ruleml.org).

[Ryan 2003] Ryan, D. J. (2003). Two views on security software liability: Let the legal system decide. *IEEE Security & Privacy*, 1(1), 70–72.

[Sackmann et al. 2006] Sackmann, S., Strüker, J., & Accorsi, R. (2006). Personalization in privacy-aware highly dynamic systems. *Commun. ACM*, 49(9), 32–38.

[Saleh et al. 2007] Saleh, M., Arasteh, A. R., Sakha, A., & Debbabi, M. (2007). Forensic Analysis of Logs: Modeling and verification. *Knowledge-Based Systems*, 20(7), 671–682.

[Schneider 2009] Schneider, F. B. (2009). Accountability for Perfection. *IEEE Security & Privacy*, 7(2), 3–4.

[Schneider 2001] Schneider, S. (2001). *The B-Method: An Introduction*. Palgrave MacMillan.

[Schneier & Kelsey 1999] Schneier, B. & Kelsey, J. (1999). Secure audit logs to support computer forensics. *ACM Trans. Inf. Syst. Secur.*, 2(2), 159–176.

[Schwarz & Mattern 1994] Schwarz, R. & Mattern, F. (1994). Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3), 149–174.

[Sen & Garg 2003] Sen, A. & Garg, V. K. (2003). Partial order trace analyzer (pota) for distributed programs. *Electr. Notes Theor. Comput. Sci.*, 89(2), 22–43.

[Servat 2007] Servat, T. (2007). Brama: A new graphic animation tool for b models. In J. Julliand & O. Kouchnarenko (Eds.), *B*, volume 4355 of *Lecture Notes in Computer Science* (pp. 274–276). Springer.

[Sidhu & Prasanna 2001] Sidhu, R. & Prasanna, V. K. (2001). Fast Regular Expression Matching Using FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (pp. 227–238).

[Sistla 1994] Sistla, A. P. (1994). Safety, liveness and fairness in temporal logic. *Formal Asp. Comput.*, 6(5), 495–512.

[Skene et al. 2007] Skene, J., Skene, A., Crampton, J., & Emmerich, W. (2007). The monitorability of service-level agreements for application-service provision. In V. Cortellessa, S. Uchitel, & D. Yankelevich (Eds.), *WOSP* (pp. 3–14). ACM.

[Stathopoulos et al. 2006] Stathopoulos, V., Kotzanikolaou, P., & Magkos, E. (2006). A framework for secure and verifiable logging in public communication networks. In J. López (Ed.), *CRITIS*, volume 4347 of *Lecture Notes in Computer Science* (pp. 273–284). Springer.

[Steer et al. 2011] Steer, S., Craipeau, N., Métrayer, D. L., Maarek, M., Potet, M.-L., & Viet Triem Tong, V. (2011). Définition des responsabilités pour les dysfonctionnements de logiciels: Cadre contractuel et outils de mise en oeuvre. In *Droit, Sciences et Techniques, Quelles Responsabilité?*, éditions Litec (Lexisnexis), collection "colloques et debats".

[Strano et al. 2009] Strano, M., Molina-Jiménez, C., & Shrivastava, S. K. (2009). Implementing a rule-based contract compliance checker. In C. Godart, N. Gronau, S. K. Sharma, & G. Canals (Eds.), *I3E*, volume 305 of *IFIP* (pp. 96–111). Springer.

[U.S. Food and Drug Administration 2011] U.S. Food and Drug Administration (2011). *LASIK Eye Surgery*. Available at: www.fda.gov/LASIK.

[Vardi 2008] Vardi, M. Y. (2008). From church and prior to psl. In O. Grumberg & H. Veith (Eds.), *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science* (pp. 150–171). Springer.

[Varol & Rotem 1981] Varol, Y. L. & Rotem, D. (1981). An algorithm to generate all topological sorting arrangements. *Comput. J.*, 24(1), 83–84.

[Vaughan et al. 2008] Vaughan, J. A., Jia, L., Mazurak, K., & Zdancewic, S. (2008). Evidence-based audit. In *CSF* (pp. 177–191). IEEE Computer Society.

[von Wright 1951] von Wright, G. H. (1951). Deontic logic. *Mind*, 60, 1–15.

[Waters et al. 2004] Waters, B. R., Balfanz, D., Durfee, G., & Smetters, D. K. (2004). Building an encrypted and searchable audit log. In *NDSS* The Internet Society.

[Wolper 1983] Wolper, P. (1983). Temporal logic can be more expressive. *Information and Control*, 56(1/2), 72–99.

[Woodcock et al. 2009] Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. S. (2009). Formal methods: Practice and experience. *ACM Computing Survey*, 41(4).

[Xu & Jeusfeld 2003] Xu, L. & Jeusfeld, M. A. (2003). Pro-active monitoring of electronic contracts. In J. Eder & M. Missikoff (Eds.), *CAiSE*, volume 2681 of *Lecture Notes in Computer Science* (pp. 584–600). Springer.

[Xu et al. 2005] Xu, L., Jeusfeld, M. A., & Grefen, P. W. P. J. (2005). Detection tests for identifying violators of multi-party contracts. *SIGecom Exchanges*, 5(3), 19–28.

[Yao-Hua Tan 2001] Yao-Hua Tan, W. T. (2001). A survey of electronic contracting related developments. In *14th Bled Electronic Commerce Conference*.