A Formal Framework for Specifying and Analyzing Logs as Electronic Evidence

Eduardo Mazza[1], Marie-Laure Potet[1], and Daniel Le Métayer[2]

[1] Verimag, Centre Équation, 2 avenue de Vignate, F-38610 Gières
{Eduardo.Mazza,Marie-Laure.Potet}@imag.fr
[2] LICIT, INRIA Grenoble Rhône-Alpes
Daniel.Le-Metayer@inrialpes.fr

Abstract. The issues of logging for determining liability requires to define, prior to a dispute, the logging system and the log analysis in a manner that would determine the parties liable for a predetermined misbehavior of the system. We propose a formal framework for specifying and reasoning about decentralized logs to be used in legal disputes. In addition, we study how previous results can be used in the incremental analysis of larger inputs to obtain precise or approximated results. We illustrate our approach with an example of a travel arrangement service.

1 Introduction

Due to the growing impact of the Information and Communication Technologies on everyday life, and the increasing complexity of computer systems, the logging of these systems raises greater challenges. Logging is a necessity for debugging a system at development time, or after a fault, for identifying security attacks, guaranteeing safety, and establishing liability for software providers. With respect to the last one, Fred B. Schneider pointed [23] that liability determination needs a mature discipline of forensics for computing systems and components. It requires to change software development practices, because, in addition to delivering systems, producers will also need to deliver instruments to show that they behave well.

The use of log as electronic evidence for establishing contractual liability is a challenging problem [18, 6, 12]. Actual solutions that propose formal models to specify liability [17, 10] are more focused on the system models rather than using logs as electronic evidences in cases of litigation. Meanwhile, other works [4, 11, 21, 3] present a well-defined model for analysing properties in logs, but a small effort has been made to specify the liability associated with the log content. Existing research stops short of discussing how one might determining whether the information found in a given log is precise enough to be used for legal disputes.

The LISE project[1] aims to address liability issues from the legal and technical points of view. The goal is to cover the whole chain of "liability engineering", from

[1] Liability Issues in Software Engineering: http://licit.inrialpes.fr/lise/

J. Davies, L. Silva, and A. Simão (Eds.): SBMF 2010, LNCS 6527, pp. 194–209, 2011.

liability specification (at contract negotiation time) to liability determination (at litigation time). With the purpose of reduce legal uncertainty, the LISE approach incorporates in B2B contracts (between services providers) an agreement about the electronic evidence to be produced and used in case of failures. According to this agreement, the parties commit to build these pieces of evidence, and to rely on them for determining their share of responsibility. We assume that certain conditions (such as, proof of authenticity and integrity) are satisfied by complementary means.

In contrast with others frameworks that focus either in liability [17, 10] or properties verification for logs [4, 11], we propose here an integrated framework that allow us to specify liability, claims, and logs as electronic evidence (Figure 1.a). As mentioned before we restrict ourselves in the context of liability defined for a contractual environment. Furthermore, we exploit the central notion of agents (seen as parties implied in a contractual engagement) that organize claims, properties and distributed logs in a very tractable way. Then, we are able to propose a general analysis procedure (Figure 1.b) allowing us to evaluate the admissibility of a given claim instance. This procedure is focused on the notion of properties attached to a given claim, rather than general properties describing the behavior of the system [4, 11, 21, 3]. The fact that, in our context, claims are defined a priori allow us to provide a simpler specification for such procedure. We also study the incremental aspects of our procedure and propose alternative solutions to obtain new results based on the results of a previous analysis.

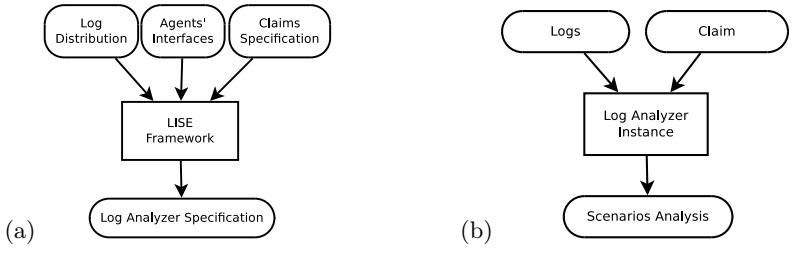

Fig. 1. Framework for liability specification and analysis

This paper is dedicated to the presentation of our framework for the formal representation and analysis of logs. We explore the aspects of the LISE methodology introduced in [15] considering the distribution and analysis of logs. Another previous work [16] studies how different distribution of logs can be classified with relation to their content w.r.t. the level of interest that logging agents have in changing the events in their logs. These can be viewed as complementary results of the present work. Section 2 introduces a motivating example and some useful notations. Section 3 presents our general model to specify logs, log distributions and claims. Section 4 introduces some classical operations on distributed logs, as extraction and merge. Finally, Section 5 gives the specification of our log analysis and its use for claim admissibility together with the studies of incremental results.

2 Case Study and Notations

In this section we present our case study and give a brief background of the B notation used in this paper.

2.1 Case Study

Throughout this paper we will use a travel booking case study as a running example of the different aspects of our framework.

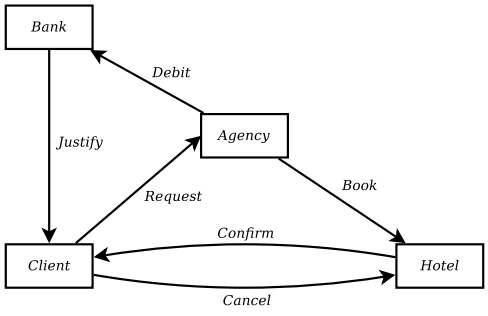

Fig. 2. Travel booking system

In this case study (Figure 2), the client (*Client*) submits a request (*Request*) for a hotel reservation to the travel agency (*Agency*). On the request arrival, *Agency* performs some research to find a hotel that supplies the specifications requested by *Client* (for example, price or hotel location). Having found a hotel, *Agency* sends a request (*Book*) to the hotel (*Hotel*) with the specification of dates and room. *Hotel* confirms the reservation sending a message ($Confirm$) to *Client*. *Client* can cancel the reservation before the reservation date free of charge. To cancel the reservation, *Client* shall send a message (*Cancel*) to *Hotel* informing that s/he no longer desires the room. If *Client* did not cancel the reservation within the time specified, *Agency* makes a debit in the client's account sending a message (*Debit*) to *Bank*. Finally, *Bank* sends a message ($Justify$) to *Client* with the justification for the payment made with his/her account.

From the *Client*'s point of view a number of things could go wrong. We consider here two claims:

Example 1. (claim $NoRoom$) Let us consider the case that *Client* submits a request and receives a justification from *Bank*, but when s/he arrives at the hotel there is no information about the reservation in *Hotel*'s registry. *Client* complains to *Agency* that s/he received a message from *Bank* and still there is no room available.

Example 2. (claim *LateCancel*) Let us consider another claim where having canceled the reservation, *Client* is still charged by *Agency*. Then, *Client* complains to *Agency* that s/he sent the message to cancel the reservation. On the other side, *Agency* complains that *Client* did not send the message before the reservation date. In this example, the time that the client sends the message can be confused with the time that the message is received by $Hotel$. It is necessary to make clear in the agreement between *Agency* and *Client* how such details will be taken into account.

In such situations, liability can be specified in terms of well-defined properties written in function of events recorded in the logs. Consider, for example, the scenario in Figure 3. In this scenario *Agency* does not send the request of reservation to $Hotel$, but still charges the client by sending a message to $Bank$. *Client* receives the justification from $Bank$ and could think that, although s/he has no confirmation from $Hotel$ (maybe *Client* does not even know that should receive a confirmation from $Hotel$), the fact that s/he was charged by *Agency* assures the confirmation of the reservation.

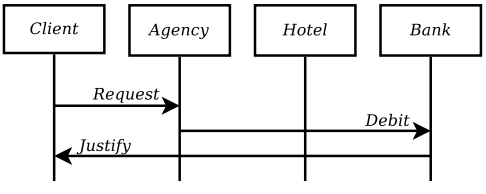

Fig. 3. Sequence diagram for possible scenario where claim $NoRoom$ will happen

For the agreement to work it is necessary that the values found in the logs can be trusted by all entities of the system and that it contains no error. In this paper, we assume that this hypothesis is valid and the values that compose the logs correspond exactly to what happened (no duplication, no replication, no loss).

2.2 Notation B

This formal framework is based on the B-method [1]. This methodology was chosen because it allows specifications that are focused both on data and behavior. Another important aspect for our lawyer partners is the industrial status of this approach [5, 2] that is a positive argument in term of trust, a central notion in law.

We give here a short summary of the B notations we use in this paper. Given two sets, A and B, the set of relations between A and B, the set of functions from A to B, and the set of partial functions from A to B are respectively denoted by $A \leftrightarrow B$, $A \rightarrow B$, and $A \nrightarrow B$. We note $\mathcal{F}(A)$ all finite subsets of A, $R[U]$ the relational image of U under the relation R (set of all elements that are under relation R with an element of U), and $union(A)$ the generalized union of A (union of all elements of the set A). We denote by f^{-1} the inverse function for f and by $iseq(A)$ the set of

injective sequences of elements of A. A sequence is delimited by the square brackets ('[' and ']') and its elements separated by commas (','). The notation $\lambda\, x.(P \mid E)$ denotes the function that maps every x that verifies the predicate P into the value of expression E. Finally, for readability, we sometimes omit the declaration of the type of variables whenever we judge the type is easy to infer.

3 Logs and Claims

In this section we present how, in our framework, we specify the expected information about the system, the logs and the claims. The proposed model is based on exchanged messages that are used to represent the interactions between the entities of the system. We will assume here that each message is unique. We will also assume that we are able to identify the origin of exchanged messages. However, our framework can be easily adapted for working without assuming such hypothesis. This model of communication is well adapted to the domain of B2B applications that we mentioned in the introduction, including electronic service and resource provision.

We consider a system consisting of agents participating in some form of interactions described as events, and each event is performed by one of the agents. Particular agents may have the ability to monitor the events performed by itself and other agents and record these events in a log[2]. The agents should agree that the produced logs are liable for representing the event's history.

3.1 System Information

The machine $SystemInfo$ specifies a set of agents ($AGENT$) and a set of actions ($ACTION$). Each action is associated with the agent that can execute it by the function $Interface$. The information associated with the system in study is marked in bold.

Example 3. (agents and interface) Let us write the machine $SystemInfo$ for our case study such that:

MACHINE $SystemInfo$
SETS
  $AGENT = \{\mathbf{Client}, \mathbf{Agency}, \mathbf{Hotel}, \mathbf{Bank}\}$;
  $ACTION = \{\mathbf{Request}, \mathbf{Book}, \mathbf{Debit}, \mathbf{Confirm}, \mathbf{Justify}, \mathbf{Cancel}\}$
CONSTANTS $Interface$
PROPERTIES
  $Interface : ACTION \rightarrow AGENT \wedge$
  $Interface = \{(\mathbf{Request}, \mathbf{Client}), (\mathbf{Book}, \mathbf{Agency}), (\mathbf{Debit}, \mathbf{Agency}),$
  $(\mathbf{Confirm}, \mathbf{Hotel}), (\mathbf{Justify}, \mathbf{Bank}), (\mathbf{Cancel}, \mathbf{Client})\}$
END

[2] Logs can also be produced by some mechanism that does not directly belong to the system, such as protocol sniffers.

The actions specified for a given system may depend on the claims that need to be evaluated. Other actions, such as confirmation of the reservation between $Hotel$ and $Agency$, were not considered here to keep the simplicity of the example.

3.2 Logs and Distribution

An event ($EVENT$) is described as a tuple consisting of the type of event (OP), the source and destination agents, and the action to be executed. A log file (LOG_FILE) is defined as a pair consisting of the set of agents that have their events logged and the sequence of events, recorded in the order of execution.

A log architecture (LOG_ARCH) represents a set of logs produced during the execution of the system. The constant $Dist$ describes how we have chosen to regroup and distribute logs. Each element $X \in Dist$ represents that a single log file records the events performed by the agents in X. Notice that in this definition the events of an agent might be recorded in more than one log file.

Logs are modeled using the machine $LogModel$.

MACHINE $LogModel$
INCLUDES $SystemInfo$
SETS $OP = \{Send, Rec\}$
CONSTANTS $EVENT$, LOG_FILE, LOG_ARCH, $Dist$
PROPERTIES
$EVENT = OP \times AGENT \times AGENT \times ACTION \wedge$
`/* Log files */`
$LOG\_FILE = \mathcal{F}(AGENT) \times \text{iseq}(EVENT) \wedge$
`/* Definition of the chosen log distribution */`
$\mathbf{Dist} \subseteq \mathcal{F}(\mathbf{AGENT}) \wedge \mathbf{Dist} = \{\cdots\} \wedge$
`/* Log architectures defined as a set of log files */`
$LOG\_ARCH = \{logs \mid logs \in \mathcal{F}(LOG\_FILE) \wedge \forall log.(log \in logs \Rightarrow$
$agents(log) \in Dist)\}$
END

In the rest of the paper we use the functions $agents$ and $content$ that maps every log file into its agents and content respectively. The function $events$ maps every log file into the set of events (rather than the sequence) of the log's content. We abuse our notation and assume that $agents$ and $events$ can also be applied to log architectures: applying $agents$ (or $events$) to a log architecture log_arch is equivalent to applying $agents$ (or $events$) to each log file that belongs to log_arch and make the union of the results.

Example 4. (log distribution) Let us consider the following two examples of distributions:

1. $Dist = \{\{\mathbf{Client}\}, \{\mathbf{Agency}\}, \{\mathbf{Hotel}\}, \{\mathbf{Bank}\}\}$
2. $Dist = \{\{\mathbf{Client}, \mathbf{Agency}\}, \{\mathbf{Hotel}\}, \{\mathbf{Bank}\}\}$

In the first example we describe a distribution where each agent is logged independently. In the second example we assume that there is a single log file that records the events performed by $Client$ and $Agency$.

3.3 Properties and Claims

In our case study, the agents make an agreement to trust the content of logs, in order to establish liability for a given set of claims. Then it is necessary to express these claims in terms of log events in an unambiguous way. We assume that liability for claims take the form: "if *Prop* holds, then agent defendant is responsible". This structure is explained in more details in [15].

The machine *Claims* below introduces some general definitions and allows us to declare particular instances of properties and claims.

MACHINE $Claims$
INCLUDES $LogModel$
CONSTANTS $PROP, CLAIM$
PROPERTIES
`/* Log Properties */`
$PROP = \{prop \mid prop \in \mathcal{F}(AGENT) \times (LOG\_FILE \nrightarrow \mathsf{BOOL}) \wedge$
$\forall(ags, log).(prop = (ags, log) \Rightarrow ags = agents(log))\} \wedge$
$\mathbf{prop_{NoRoom}} \in \mathbf{PROP} \wedge \mathbf{prop_{LateCancel}} \in \mathbf{PROP} \wedge \ldots \wedge$
`/* Claims */`
$CLAIM = \{claim \mid claim \in (AGENT \times AGENT \times PROP) \wedge$
$\forall(plain, def, prop).(claim = (plain, def, prop) \Rightarrow$
$\{plain, def\} \subseteq agents(prop))\} \wedge$
$\mathbf{NoRoom} \in \mathbf{CLAIM} \wedge \mathbf{NoRoom} = (\mathbf{Client}, \mathbf{Agency}, \mathbf{prop_{NoRoom}}) \wedge$
$\mathbf{LateCancel} \in \mathbf{CLAIM} \wedge \mathbf{LateCancel} = (\mathbf{Client}, \mathbf{Agency}, \mathbf{prop_{LateCancel}})$
END

Due to the distributed nature of logs, the first element of a property explicitly states the agents that are concerned with this property. The second element of the property is a partial function that maps a log file to true or false depending if the log file verifies the property. We use the notation $agents(prop)$ and $val(prop)$ indicating respectively the first and second parts of a property *prop*. In a property it is imposed that the domain of logs matches exactly with the agents concerned by the property. Claims are tuples consisting of a plaintiff, a defendant, and a property that describes the claim, with the plaintiff and the defendant belonging to the agents of the property.

Example 5. (property $prop_{NoRoom}$) Now, consider the claim $NoRoom$ described in Section 2.1. Let $prop_{NoRoom}$ be the property that describes the claim where *Client* complains of *Agency* being responsible for charging him/her without booking the reservation:

$$\begin{gathered} agents(prop_{NoRoom}) = \{Client, Agency\} \wedge \\ val(prop_{NoRoom}) = \lambda\, log.(agents(log) = \{Client, Agency\} \mid \\ (Send, Client, Agency, Request) \in events(log) \wedge \\ (Send, Agency, Bank, Debit) \in events(log) \wedge \\ (Send, Agency, Hotel, Book) \notin events(log)) \wedge \\ pos((Send, Client, Agency, Request), log) < \\ pos((Send, Agency, Bank, Debit), log)) \end{gathered}$$

This definition starts limiting the agents of the property to *Client* and *Agency*. The second part of the definition gives the conditions that verify the property. We use the function $pos(ev, log)$ that maps every event ev and log log such that $ev \in events(log)$, into the position of ev within the sequence $content(log)$.

4 Log Functions

Like others frameworks dedicated to distributed systems we provide some functions for manipulating distributed logs. The functions presented here are based in the well known relation "happened-before" introduced in the early work of Lamport [14].

4.1 Log Extraction

The function *extract* allow us to obtain information contained in a log file concerning a certain group of agents.

Definition 1. (function *extract*) *The partial function extract* : $(\mathcal{F}(AGENT) \times LOG\_FILE) \nrightarrow LOG\_FILE$ *maps every pair* (ags, log) *such that* $ags \subseteq agents(log)$*, into* log_{ext} *with* log_{ext} *having the following properties:*

1. $agents(log_{ext}) = agents(log)$
2. $events(log_{ext}) = \{ev \mid ev \in events(log) \wedge \exists(ag_1, ag_2, ac).(ag_1 \in agents(log) \wedge ev = (Send, ag_1, ag_2, ac) \vee ev = (Rec, ag_2, ag_1, ac))\}$
3. $\forall(ev_A, ev_B).(ev_A \in events(log_{ext}) \wedge ev_B \in events(log_{ext}) \wedge pos(ev_A, log_{ext}) < pos(ev_B, log_{ext}) \Rightarrow pos(ev_A, log) < pos(ev_B, log))$

The last two properties of Definition 1 respectively state that the extracted log (*2.*) contains all events that represent messages sent or received by the agent in ags and (*3.*) respects the order of events in log. In the rest of this paper we use $extract_{ags}(log)$ to denote $extract(ags, log)$.

Example 6. (application of *extract*) Let us imagine the variable $log \in LOG\_FILE$ representing the log of *Client* and *Agency* for the scenario described in Figure 3:

$$log = (\{Client, Agency\}, [(Send, Client, Agency, Request), (Rec, Client, Agency, Request), (Send, Agency, Bank, Debit), (Rec, Bank, Client, Justify)])$$

Then, we can use *extract* to obtain the events performed only by *Agency*:

$$extract_{\{Agency\}}(log) = (\{Agency\}, [(Rec, Client, Agency, Request), (Send, Agency, Bank, Debit)]).$$

4.2 Log Merge

The relation *merge* produces, for a given log architecture, the set of logs respecting the order of each log file in the architecture and the order of corresponding *Send*/*Rec* events. Each log produced by this relation represents a scenario describing the sequence of events in the order they were performed.

Definition 2. (relation *merge*) *The relation merge* : $LOG\_ARCH \leftrightarrow LOG\_FILE$ *maps every log architecture into logs that will represent all possible total orders for this architecture. That is, for any log_arch and log, such that* $(log\_arch, log) \in merge$*, then:*

1. $agents(log) = agents(log\_arch)$
2. $events(log) = events(log\_arch)$
3. $\forall log_{ag} \in log\_arch \Rightarrow extract_{agents(log_{ag})}(log) = log_{ag}$
4. $\forall(ev_B, ag_1, ag_2, ac).(ev_B \in events(log\_arch) \wedge ev_B = (Rec, ag_1, ag_2, ac) \wedge ag_1 \in agents(log\_arch) \Rightarrow \exists(ev_A).(ev_A \in events(log\_arch) \wedge ev_A = (Send, ag_1, ag_2, ac) \wedge pos(ev_A, log) < pos(ev_B, log))$

The last two properties in Definition 2 respectively state that the scenario represented by *log* (*3.*) respects the local order of each log file in *log_arch* and (*4.*) for every event of type *Rec* if there is a corresponding event of type *Send* then the event of type *Send* precedes the event of type *Rec* in this scenario. That is, the relation *merge* collects all interleaving that respect local and causal orderings.

Example 7. (application of *merge*) Let us, from now on, denote events omitting the sender and receiver and using the simplified notation $(Send, ac)$ to represent an event of the form $(Send, ag_1, ag_2, ac)$ and similar for (Rec, ac). Let us imagine the distribution *log_arch* composed by the logs of *Client* and *Agency* where:

$$\begin{aligned} log_{Client} &= (\{Client\},\ [(Send, Request), (Rec, Justify)]) \\ log_{Agency} &= (\{Agency\}, [(Rec, Request), (Send, Debit)]) \end{aligned}$$

Then, $merge[log\_arch]$[3] produces the set $\{log_1, log_2, log_3\}$ such that:

$$\begin{aligned} log_1 &= \{Client, Agency\}, [(Send, Request), \\ &(Rec, Justify), (Rec, Request), (Send, Debit)] \\ log_2 &= \{Client, Agency\}, [(Send, Request), \\ &(Rec, Request), (Rec, Justify), (Send, Debit)] \\ log_3 &= \{Client, Agency\}, [(Send, Request), \\ &(Rec, Request), (Send, Debit)], (Rec, Justify) \end{aligned}$$

That is, the event $(Rec, Justify)$ can be permuted with the events $(Rec, Request)$ and $(Send, Debit)$ because there is no order constraint between these events. However, we know that all these events shall come after the event $(Send, Request)$ and that the event $(Send, Debit)$ comes after the event $(Rec, Request)$.

[3] In this paper we use the notation $merge[log\_arch]$ rather than $merge[\{log\_arch\}]$ for sake of simplicity.

5 Log Analysis

In this section, we give the general method of our log analyzer. Given the values of the current logs and a particular claim, the aim is to establish the truthfulness of this claim based on the content of the logs.

5.1 Log Analyzer

First, we present the definition of the operation *PropAnalysis*.

$scen, ok \leftarrow PropAnalysis(logs, prop) \hat{=}$
  PRE
    $logs \in LOG\_ARCH \wedge prop \in PROP \wedge agents(prop) \subseteq agents(logs)$
  THEN
    $scen := extract_{agents(prop)}[merge[logs]]$;
    $ok := scen \cap val(prop)^{-1}[\{\mathsf{TRUE}\}]$
  END
END

For a given set of logs *logs*, the operation *PropAnalysis* researches scenarios that hold for a given property *prop*. Two results are computed: the set of possible different scenarios (*scen*) zooming only on the concerned agents of the property, and among them, the subset of these scenarios that fulfill the property (*ok*). The ratio between the two sets *scen* and *ok* may inform us the level truthfulness of the researched property for the given logs.

Now, the method for the analysis of a claim of the form $(plain, def, prop)$ proceeds as follows:

1. Select a set of logs in the global log architecture such that $\{plain, def\} \subseteq agents(prop)$. These *logs* may represent the current global set of logs or only a subset of them.
2. Execute $scen, ok \leftarrow PropAnalysis(logs, prop)$
3. Analyze the results in term of the admissibility of the claim and its explanation. For instance:
   - if $ok = scen$ the property holds for all scenarios, which leads us to conclude that the claim is valid and the defendant is responsible;
   - if $ok = \emptyset$ the property is false for all scenarios, and leads us to conclude that the claims should be rejected;
   - otherwise, an appropriate examination of scenarios that fulfill (or not) the property has to be conducted.

If the results are not conclusive, it could be necessary to increase the amount of logs used in the analysis, as we explain in Section 5.2.

Example 8. (claim *NoRoom* analysis) Suppose we have the following current logs representing the scenario illustrated in Figure 3:

$$\begin{aligned} log_{Client} &= (\{Client\}, \ [(Send, Request), (Rec, Justify)]) \\ log_{Agency} &= (\{Agency\}, [(Rec, Request), (Send, Debit)]) \\ log_{Bank} &= (\{Bank\}, \ [(Rec, Debit), (Send, Justify)]) \\ log_{Hotel} &= (\{Hotel\}, \ [\,]) \end{aligned}$$

If we execute *PropAnalysis* for *NoRoom* using all these logs it is clear that *Agency* is liable for *Client*'s damages[4], because only one scenario is generated and fulfills the property $prop_{NoRoom}$. However, suppose that obtaining log_{Bank} can be an issue and that we want to verify if *Agency* is responsible for the incident. Then, the property can be analyzed for a reduced architecture using only log_{Client} and log_{Agency}. This setting will generate the three scenarios described in Example 7, but for all of them $prop_{NoRoom}$ holds, which leads to the conclusion that the claim is valid and *Agency* should be responsible.

5.2 Incremental Analysis

When results are not precise enough, a deeper investigation can be conducted, for instance, inspecting more logs or in looking for some other dysfunctions. We focus now on an incremental approach to obtain more precise results when more logs are later added in the analysis.

Let *logs* and *prop* be respectively the logs and the property that have been already analyzed. Now let $logs'$ be a new set of logs, selected in order to obtain more precise results. Here we study how $scen', ok' \leftarrow PropAnalysis(logs \cup logs', prop)$ can be incrementally computed, reusing the results obtained by $scen, ok \leftarrow PropAnalysis(logs, prop)$.

Incremental calculus for $merge[logs \cup logs']$. The following property allows us to compute $merge[logs \cup logs']$.

Property 1. (merge by parts) Let *logs* and $logs'$ be two sets of logs. We have:

$$merge[logs \cup logs'] = \text{union}(\{merge[logs' \cup \{log\}) \mid log \in merge[logs]\}]$$

Then, in the first step of $PropAnalysis(logs \cup logs', prop)$ it is possible to reuse the result of $merge[logs]$ to compute $merge[logs \cup logs']$. However, the operation *extract* does not distribute on the operator merge (see Property 2 below).

Incremental calculus for $scen'$ and ok'. The results of $PropAnalysis(logs \cup logs', prop)$ can be approximated using the previous results for *scen* and *ok* in the following way:

[4] In law, the term "damage" is associated with an award of money to be paid as compensation for loss or injury. In this paper we refer to this term in a more general meaning.

$iscen, iok \leftarrow IncrPropAnalysis(logs', prop, scen, ok) \hat{=}$
PRE
$logs' \in LOG\_ARCH \wedge prop \in PROP \wedge$
$scen \in \mathcal{F}(LOG\_FILE) \wedge ok \in \mathcal{F}(LOG\_FILE) \wedge$
$agents(prop) \subseteq agents(scen) \wedge$
$agents(prop) \subseteq agents(ok)$
THEN
$iscen := extract_{agents(prop)}[\text{union}(\{merge[logs' \cup \{log\}] \mid log \in scen\})];$
$iok := extract_{agents(prop)}[\text{union}(\{merge[logs' \cup \{log\}] \mid log \in ok\})]$
END
END

Now we can compare:

$$\begin{aligned} scen,\ ok &\leftarrow PropAnalysis(logs, prop) \\ scen',\ ok' &\leftarrow PropAnalysis(logs \cup\ logs', prop) \\ iscen, iok &\leftarrow IncrPropAnalysis(logs', prop, scen, ok) \end{aligned}$$

The result is:

$$\boxed{ok' \subseteq iok \subseteq ok \text{ and } scen' \subseteq iscen \subseteq scen}$$

This result can be concluded using the following property.

Property 2. (extraction by parts) Let log be a log, log_arch a log architecture, and ags a set of agents. We have:

$$\begin{gathered} extract_{ags}[merge[log\_arch \cup \{log\}]] \subseteq \\ extract_{ags}[merge[log\_arch \cup \{extract_{ags}(log)\}]] \end{gathered}$$

This property suggests that when an extraction is made before the merge some information about the order of events may be lost and may result in a large set of scenarios for $merge$. Since $IncrPropAnalysis$ uses the results $scen$ and ok of $PropAnalysis$, some information may be lost with the extraction of the events that only concern the agents of the property.

Application. Although we do not obtain exact results, this operation is interesting because it does not retest the researched property, what can be a complex step. Moreover in some cases we may obtain conclusive results. For instance, it is always the case where $iok = \emptyset$ or $iok = iscen$.

Example 9. Suppose the claim $LateCancel = (Client, Agency, prop_{LateCancel})$ such that:

$$\begin{gathered} agents(prop_{LateCancel}) = \{Client, Agency\} \wedge \\ val(prop_{LateCancel}) = \lambda\ log.(agents(log) = \{Client, Agency\} \mid \\ (Send, Cancel) \in events(log) \wedge \\ (Send, Debit) \in events(log) \wedge \\ pos((Send, Cancel), log) < pos((Send, Debit), log)) \end{gathered}$$

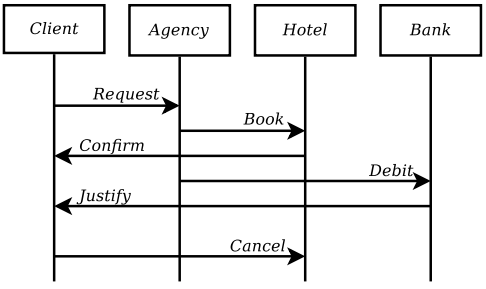

Fig. 4. Sequence diagram for possible scenario

That is, *Agency* is responsible for the claim *LateCancel* if *Client* sends a message to cancel the reservation before *Agency* sends the message to *Bank* charging the client. Now, consider the scenario described in Figure 4 and an initial call of *PropAnalysis* only with log_{Client} and log_{Agency}. The values for the logs can be written as follows:

$$\begin{aligned} log_{Client} &= (\{Client\}, \ [(Send, Request), (Rec, Confirm), (Rec, Justify), \\ &\qquad (Send, Cancel)]) \\ log_{Agency} &= (\{Agency\}, [(Send, Request), (Send, Book), (Send, Debit)]) \end{aligned}$$

Then, it is not possible to verify if the claim is valid, because the result of $merge[\{log_{Client}, log_{Agency}\}]$ will produce 20 different scenarios where in ten of them the position of $Cancel_{Send}$ is before $Debit_{Send}$.

Now, we call *IncrPropAnalysis* with the result of the previous analysis and log_{Bank} that should contain the following value:

$$log_{Bank} = (\{Bank\}, [(Rec, Debit), (Send, Justify)])$$

The bank's log adds a restriction to the previous scenarios. Now we know that *Client* tries to cancel the reservation after the message $(Rec, Debit)$. After execute *IncrPropAnalysis* we remain with four scenarios in *iscen* because we are not sure about the position of the event $(Rec, Confirm)$. However, we obtain $iok = \emptyset$ and can conclude that the claim should be rejected.

6 Conclusion

This paper has presented some parts of the LISE context [15], a project with the objective to create a formal framework to precisely define liability in IT systems and establish liability in case of failure. Our framework provides a model for formally specifying the aspects of liability in a contractual setting. We also present the specification for a log analyzer that can be used to establish the validity of claims with relation to a given distributed log architecture. Finally, we explore the aspects of incremental analysis for this log analyzer.

6.1 Related Works

A large amount of work has been dedicated to the formal specification of contracts, among them [13, 9, 8]. Such specifications are useful but they still lack in specific issues of legal evidences and can be considered as a complementary approach to our framework.

Works in the domain of forensics [19, 20, 22, 26] and audit [24, 7, 27] make general references to analysis of digital information in a legal setting. However, in general these contributions seem to be targeted towards security issues and attack detection rather than define liability for possible claims that may rise.

The contributions presented in this paper differs from other works such as [25] and [7]. In these works the authors make reference to liability using logs but they are more focused in the aspects of monitorability of events and how the logs should be produced. The management of log distributions related with liability is an important related topic which has been covered by our prior work [16], where we characterized the acceptability of a given distribution with respect to a trust relationship and the neutrality of agents to log some given events.

6.2 Future Work

In this paper, we assumed that logs will not contain inconsistencies or incorrect values. As commented before, this hypothesis is justified by the use of other means, for instance digital signatures, that will assure characteristics such as integrity and non-repudiation for the logs. Another possibility, to verify log integrity, is the use of redundancy in a log architecture, when the events concerning one agent is recorded several times. However, situations where this hypothesis is not assumed has been considered in our previous works [16].

In the future we will extend our framework to take into account parameterized claims and properties (for instance to extend our case study with several clients and hotels). Future work also includes the integration of our work presented in [16] in this general framework.

Acknowledgement

This contribution is part of the LISE project (ANR-07-SESU-007) funded by ANR.

References

[1] Abrial, J.: The B-Book. Cambridge University Press, Cambridge (1996)
[2] Badeau, F., Amelot, A.: Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 334–354. Springer, Heidelberg (2005)
[3] Barringer, H., Groce, A., Havelund, K., Smith, M.H.: An Entry Point for Formal Methods: Specification and Analysis of Event Logs. CoRR abs/1003.1682 (2010)

[4] Barringer, H., Groce, A., Havelund, K., Smith, M.H.: Formal Analysis of Log Files. Aerospace Computing, Information, and Communication (to appear, 2010)
[5] Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: A Successful Application of B in a Large Project. In: Woodcock, J.C.P., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999)
[6] Buskirk, E.V., Liu, V.T.: Digital Evidence: Challenging the Presumption of Reliability. Journal of Digital Forensic Practice 1(1), 19–26 (2006)
[7] Cederquist, J.G., Corin, R., Dekker, M.A.C., Etalle, S., den Hartog, J.I., Lenzini, G.: Audit-based Compliance Control. International Journal of Information Security 6(2-3), 133–151 (2007)
[8] Farrell, A.D.H., Sergot, M.J., Sallé, M., Bartolini, C.: Using the Event Calculus for Tracking the Normative State of Contracts. International Journal of Cooperative Information Systems (IJCIS) 14(2-3), 99–129 (2005)
[9] Fenech, S., Pace, G.J., Schneider, G.: CLAN: A tool for contract analysis and conflict discovery. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 90–96. Springer, Heidelberg (2009)
[10] Grossi, D., Royakkers, L.M.M., Dignum, F.: Organizational Structure and Responsibility. Artificial Intelligence and Law 15(3), 223–249 (2007)
[11] Hallal, H., Boroday, S., Petrenko, A., Ulrich, A.: A Formal Approach to Property Testing in Causally Consistent Distributed Traces. Formal Aspects of Computing 18(1), 63–83 (2006)
[12] Insa, F.: The Admissibility of Electronic Evidence in Court (AEEC): Fighting against High-Tech Crime - Results of a European Study. Journal of Digital Forensic Practice 1(4), 285–289 (2006)
[13] Kyas, M., Prisacariu, C., Schneider, G.: Run-Time Monitoring of Electronic Contracts. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 397–407. Springer, Heidelberg (2008)
[14] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM 21(7), 558–565 (1978)
[15] Le Métayer, D., Maarek, M., Mazza, E., Potet, M.L., Viet Triem Tong, V., Craipeau, N., Frénot, S., Hardouin, R.: 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 (2010)
[16] Le Métayer, D., Mazza, E., Potet, M.L.: Designing Log Architecture For Legal Evidence. In: Software Engineering And Formal Methods (SEFM). IEEE, Los Alamitos (2010)
[17] Lima, T.D., Royakkers, L.M.M., Dignum, F.: A Logic For Reasoning About Responsibility. Logic Journal of the IGPL 18(1), 99–117 (2010)
[18] Maurer, U.M.: New Approaches to Digital Evidence. Proceedings of the IEEE 92(6), 933–947 (2004)
[19] Peisert, S., Bishop, M., Karin, S., Marzullo, K.: Toward Models for Forensic Analysis. In: Systematic Approaches to Digital Forensic Engineering, pp. 3–15. IEEE, Los Alamitos (2007)
[20] Rekhis, S., Krichène, J., Boudriga, N.: Cognitive-Maps Based Investigation of Digital Security Incidents. In: Systematic Approaches to Digital Forensic Engineering, pp. 25–40 (2008)
[21] Saleh, M., Arasteh, A.R., Sakha, A., Debbabi, M.: Forensic Analysis of Logs: Modeling and verification. Knowledge-Based Systems 20(7), 671–682 (2007)
[22] Sandler, D., Derr, K., Crosby, S.A., Wallach, D.S.: Finding the Evidence in Tamper-Evident Logs. In: Systematic Approaches to Digital Forensic Engineering, pp. 69–75 (2008)

[23] Schneider, F.B.: Accountability for Perfection. IEEE Security & Privacy 7(2), 3–4 (2009)
[24] Schneier, B., Kelsey, J.: Secure audit logs to support computer forensics. ACM Transactions on Information and System Security 2(2), 159–176 (1999)
[25] Skene, J., Raimondi, F., Emmerich, W.: Service-Level Agreements for Electronic Services. IEEE Transactions on Software Engineering 36(2), 288–304 (2010)
[26] Stirewalt, R.E.K., Dillon, L.K., Kraemer, E.: The Inference Validity Problem in Legal Discovery. In: International Conference on Software Engineering (ICSE), pp. 303–306. IEEE, Los Alamitos (2009)
[27] Waters, B.R., Balfanz, D., Durfee, G., Smetters, D.K.: Building an Encrypted and Searchable Audit Log. In: Proceedings of the Network and Distributed System Security. The Internet Society (2004)