A Multi-Modal Framework for Achieving Accountability in Multi-Agent Systems

Simon Kramer* and Andrey Rybalchenko

1 University of Tsukuba, Japan
   simon.kramer@h3.epfl.ch
2 Technical University Munich, Germany
   rybal@in.tum.de

Abstract. We present a multi-modal, model-theoretic framework for achieving accountability in multi-agent systems through formal proof. Our framework provides modalities for knowledge, provability, and time. With these modalities, we formalise the two main aspects of accountability, which are: soundness (accountability proper), i.e., for correct agents, the provability of their correctness by themselves; and completeness (auditability), i.e., for faulty agents, the eventual provability of their faultiness by others. In our framework, the accountability proof of a particular system is reduced to the proof of a few key lemmata, which the system designer needs to establish for a considered system.

Keywords
accountability (including auditability, liability, and non-repudiation); modal logics of knowledge, provability, and time; dependable distributed or multi-agent systems; Popper’s critical rationalism; Russell’s paradox.

1 Introduction

The subject matter of this paper is accountability in multi-agent or distributed systems [23], i.e., the possibility of enforcing responsibility for illegitimate or even illegal (in)action (in)effectuated by faulty agents in those systems. In plainer words, accountability allows to place blame [26] with all faulty agents (completeness aspect), and only with those agents (soundness aspect). Note that when faultiness implies illegality, accountability implies liability. In the present section, we introduce the motivation for our subject matter, the goal to be achieved in the matter, and the methodology that we employ to meet our goal.

Motivation In [28], accountability is promoted as a first-class design principle for dependable distributed systems. According to these authors’ manifesto,
“conventional techniques for dependable systems design are insufficient to de-
fend against an adversary that manipulates the system covertly in order to lie,
cheat, or steal”. According to [28], conventional techniques are insufficient due to
the increasing integration of system functionality across different trust domains
(cf. [20, 19] for a formal definition of trust domains). Within and across such
domains, abuse of trust must be not only detectable (i.e., knowable), but also
provable, in order to protect agents who behave correctly (the correct agents) from agents who do not (the faulty agents). Provability protects correct agents
from faulty agents because provability is a necessary and sufficient condition for
the non-repudiation of faulty behaviour: if you have a proof that I (in)effectuated
an illegitimate (in)action then I cannot repudiate having (in)effectuated that
(in)action, and vice versa. Note that the provability of a state of affairs is strictly
stronger than the knowledge that that state is the case. For example, an agent
may know that a certain state of affairs is the case from observation, yet not be
able to prove her knowledge to the non-observers (e.g., a judge) for lack of suf-
ficient evidence (i.e., proof). Conversely, correct agents should be able to prove
their correct behaviour to the other agents (e.g., in order to protect themselves
from corrupt auditors). Note that the creation of an accountable Internet has
been proposed as a national goal for cyberspace [22].

Goal The authors of [28] argue for the need of “new ways of thinking about
dependability and new methodologies to build and evaluate dependable systems”. Thereby, “[t]he key challenge is to develop general and practical methodologies
and techniques that can approach the ideal of full accountability for networked
systems”. Our ambition is to take up and meet this challenge. In the present
paper, our goal is to distil the declarative essence of accountability, and to deliver
a formal framework for achieving accountability in distributed or multi-agent
systems through the ideal of formal proof.

Contribution This paper applies the formal methodology of multi-agent sys-
tems [25] to a real-world distributed system supposed to guarantee accountabil-
ity. Being application-driven, we focus on the application of the methodology
rather than the meta-logical study of our (mostly standard) framework within
which we illustrate our methodology. Further, when applying the methodology,
we will focus on semantic proof, in order to make explicit the intuitive reasoning
content rather than the automatisable, computational content of accountability.
The formal treatment of accountability is a recent research topic; it is therefore
prudent to clarify our intuitions of and reasoning about accountability before we
try to automatise our reasoning about it with computers.

More precisely, our contribution is five-fold (cf. Section 3.2 for related work):

1. a general, declarative definition of accountability that achieves the ideal of a
formal transcription of the original, natural-language formulation from the
distributed systems community (cf. Sections 2.2–2.2)
2. the isolation and formalisation of three logically sufficient and modelling-wise necessary conditions under which distributed systems (as modelled in our framework) are indeed accountable (cf. Section 2.2)

3. a flexible formal framework for achieving accountability in distributed systems (cf. Section 2.1) that provides:
   (a) powerful descriptive idioms in the form of three primitives for logs (introduced here) and multiple modal operators (standard or introduced in [18])
   (b) an intuitive semantic setting for developing succinct formal proofs

4. a generic pre-establishment of accountability (cf. Section 2.2) that allows the proof for any candidate system to be:
   (a) factored into the contingent (i.e., application-specific) and the logical (i.e., conceptual) content
   (b) reduced to the proof of a few key lemmata

5. a principled case study as an illustrative example of how to apply our framework to a real-world system (cf. Section B).

Methodology In order to meet our goal, we formalise accountability—and the assumptions based on which it can be established—in a multi-modal language of a model-theoretic framework. The key concept for our formalisations is a modal notion of agent-based provability. This methodology will yield the general and declarative definition of accountability and the flexibility of the corresponding framework that we seek.

2 Accountability

In this section, we present our multi-modal framework for achieving accountability in distributed systems through formal proof. The framework provides modalities for knowledge, provability, and time. Knowledge will be used in the definition of provability; and provability, like also time, will be used in the formalisation of the two main aspects of accountability. According to the distributed-systems community [28], these aspects are: accountability soundness (accountability proper), i.e., for correct agents, the provability of their correctness by themselves. Accountability completeness (auditability), i.e., for faulty agents, the eventual provability of their faultiness by others. We recall that agent correctness (actually dually, faultiness) is a fundamental, primitive notion for distributed systems, whose guarantees are conditioned on this notion [23]. (A typical condition is that there be a minimal number of correct agents.)

2.1 Framework

We define our multi-modal framework model-theoretically for the sake of greater flexibility of modelling and proving in the framework as well as extending it. (Recall that in logic, “model-theoretic” means “set-theoretic”.) More precisely:
1. when modelling in our framework, we may exploit the *definitional power* of our set-theoretic semantic setting, e.g., for the application-specific definition of agent correctness (cf. Appendix B.3 for a functional example).

2. For proving in our framework and when extending it, we need not worry about:
   - (a) difficult or even impossible completeness proofs of axiomatisations because our framework is set up in the already axiomatised set-theoretic setting.
   - (b) constraining decidability and complexity issues, which, when respected, could limit the expressiveness and thus applicability of our framework.

3. When proving (semantically) in our framework, we may fully exploit the:
   - (a) *descriptive power* of our modal operators, which are logical formalisations of frequently needed, intuitive natural-language idioms.
   - (b) *deductive power* of the deduction theorem\(^3\), which we could not when proving syntactically because axiomatisations of knowledge and time (which are concepts to which we need appeal) do not enjoy the (global) deduction theorem [10].

With our framework, we get the advantages of the deductive world of (semantic) proofs and the descriptive world of modal operators, *without* the disadvantages of either world, namely the need for completeness proofs and the absence of a (global) deduction theorem for most modal proof systems.

---

**Communication model** For the sake of the faithful modelling of distributed systems, we choose *message passing* (as opposed to shared-variable communication) as the underlying communication model of our framework. For that, we define the following generic *message language*.

**Definition 1 (Message language and derivation).** Let \( \mathcal{A} \) designate an arbitrary finite set of unique agent names\(^4\) \( a, b, c \) etc. Then,

\[
\mathcal{M} \ni M ::= B \mid a \mid [M] \mid [M]_a \mid (M, M) \mid "S"
\]

defines the set \( \mathcal{M} \) of message terms \( M \) with application-specific base data \( B \), agent names\(^5\) \( a \), message hashes \( [M] \), signed messages \( [M]_a \), message pairs \( (M, M) \), and quoted pieces of syntax \( S \) such as message-carrying events (see below for their definition and Section B for an example of their quoted use in log messages).

Further,

\[
\mathcal{L} \subseteq \mathcal{M}
\]

\(^3\) The property of a proof system that implications can be proven from the conclusion of their consequent under the hypothesis of their antecedent.

\(^4\) i.e., agent names injectively map to agents (here, names are identifiers).

\(^5\) Agents are referred to by their (unique) name, which are transmittable data, i.e., messages. Agents can be network nodes, parties, processes, processors, real or virtual machines, users, etc.
designates a set of application-specific logs \( L \), to be determined by the considered application (cf. Section B for an example).

Furthermore, \( \vdash_a \subseteq 2^M \times M \) designates a relation of message derivation à la Dolev-Yao \[8\] for agent \( a \in A \) such that for all \( D \subseteq M \):

- \( D \cup \{ M \} \vdash_a M \) (the trivial derivation)
- for all \( b \in A \), \( D \vdash_a b \) (agent names are guessable)
- if \( D \vdash_a M \) then \( D \vdash_a [M]_a \) (hashing)
- if \( D \vdash_a M \) then \( D \vdash_a [M]_a \) (personal signature synthesis)
- for all \( b \in A \), if \( D \vdash_a [M]_a \) then \( D \vdash_a M \) (universal signature “analysis” and message recovery)
- \( (D \vdash_a M \text{ and } D \vdash_a M') \) iff \( D \vdash_a (M, M') \) (pairing).

Notice that \( M \) is generic in the application-specific base data \( B \) and the quoted pieces of syntax “\( S \)”, which can be freely instantiated in the considered application. Further, observe that we assume the existence of an unforgeable mechanism for signing messages, which we model with the above signature synthesis and “analysis” rules. In trusted distributed systems, such a mechanism is trivially given by the inclusion of the sender’s name in the sent message. In distrusted distributed systems, such a mechanism can be implemented with classical certificate-based or directly with identity-based \[16\] public-key cryptography. We also assume the existence of a mechanism for hashing messages. According to \[28\], “[s]ecure hashes and digital signatures are fundamental building blocks for accountable systems”.

**Setup** We subsequently define our (mostly standard) framework, which consists of a logical language with a model-theoretic semantics. The logical language provides the following, particularly notable (epistemic) constructions:

- A relational symbol \( \text{knows} \) for *individual knowledge*, which is an instance of knowledge in the sense of the transitive use of the verb ‘to know’, here ‘to know a message à la Dolev-Yao’ \[8\]. Individual knowledge will be modelled with the relation \( \vdash_a \) of message derivation (cf. Definition 1).
- A standard constructor \( \text{knows} \text{ that} \) for *propositional knowledge*, which is an instance of knowledge in the sense of the use of the verb ‘to know’ with a clause, (i.e., to know that a statement is true). Propositional knowledge will be modelled with an *indistinguishability relation* between system states.
- For a monograph on propositional knowledge, see \[9\].
- A constructor \( \text{knows} \text{ if} \) for a non-trivial combination of the two previous kinds of knowledge that expresses propositional knowledge *conditioned on* the individual knowledge of \( M \). This conditioning will be modelled as a hypothetical message reception event (e.g., from an oracle) that is inserted at the current moment in time (and thus added to the agent’s individual knowledge). For a detailed exposition, see \[18\].
- A standard constructor \( \text{knows} \text{ it is common knowledge} \) for a *collective* kind of propositional knowledge.Informally, a statement \( \phi \) is common knowledge when all agents know that \( \phi \) is true (call this new statement \( \phi' \)), all
agents know that $\phi'$ is true (call this new statement $\phi''$), all agents know that $\phi''$ is true (call this new statement $\phi'''$), etc. Note that depending on the properties of the employed communication lines, common knowledge may have to be pre-established off those lines along other lines. For a detailed treatment of common knowledge in distributed systems, see [14].

The formal definitions for all this knowledge follow now.

**Definition 2 (Framework).** Let $\Phi$ designate our logical language of closed formulae $\phi$ as defined in Table 1. There, $\phi$ denotes corresponding unary open formulae with a single free term variable $m$. Notice that the formulae above the dashed line are atomic (for elementary facts), and those below compound (formed with operators). Also, note that all operators except $K_M^a$, which was introduced in essentially the same form in [18], are standard [25]. Then, given the set

$$E \ni \varepsilon := S(a, M, b) \quad \text{“a sends M to b”}$$

$$\mid R(a, M) \quad \text{“a receives M”}$$

of system events $\varepsilon$ (also noted as $\varepsilon(a)$) for message sending and receiving, and the set $E^\omega (E^\omega)$ of (in)finite traces over $E$, we define the satisfaction relation $|= \subseteq (E^\omega \times \mathbb{N}) \times \Phi$ of our framework in Table 2. There,

- “iff” abbreviates “by definition, if and only if”
- $E[i]$ designates the system event (say $\varepsilon$) of the infinite trace $E \in E^\omega$ inspected at the position $i \in \mathbb{N}$
- $\text{msg}(\varepsilon) := \{M, \varepsilon\}$ where $\varepsilon \in \{S(a, M, b), R(a, M)\}$ designates a function for message extraction, which we tacitly lift from events $\varepsilon \in E$ to finite traces thereof (Notice the use of quoted syntax, cf. Definition 1!)
- $E[i]$ (resp. $E[i]$) designates the finite prefix trace up to (resp. infinite suffix trace from) and including the event at the position $i \in \mathbb{N}$ of the infinite trace $E \in E^\omega$
- $\downarrow : (E^\omega \times A) \rightarrow E^\omega$ designates a projection function projecting a finite trace onto an agent’s (local) view such that for all $E \in E^\omega$ and $a \in A$,

$$\varepsilon \downarrow a := \varepsilon \quad \text{(the empty trace)}$$

$$\{\varepsilon \cdot E \downarrow a \} := \begin{cases} \varepsilon \cdot (E \downarrow a) & \text{if } \varepsilon \in \{S(a, M, b), R(a, M)\} \\ E \downarrow a & \text{otherwise} \end{cases}$$

- $\text{logs} : A \rightarrow 2^E$ designates an application-specific log selection function selecting the set of well-formed logs for a given agent, to be defined by the considered application (cf. Section B for an example)
- $\llbracket : \mathcal{L} \rightarrow E^\omega$ designates a log-to-trace transcription (or de-quotient) function (again to be defined by the considered application)
- $\uparrow : (E^\omega \times A) \rightarrow E^\omega$ designates a completion function completing finite traces with an infinite suffix of arbitrary events beyond the agent’s view (for well-definedness only, i.e., conceptually inessential)
\[ \Phi \ni \phi \quad ::= \quad \text{correct}(a) \quad \text{“a is correct”} \]
\[ \text{sends}(a, M, b) \quad \text{“a sends M to b”} \]
\[ \text{a receives M} \quad \text{“a receives M”} \]
\[ \text{a } \alpha \text{ M} \quad \text{“a knows M”} \]
\[ M \text{ wfLog } a \quad \text{“M is a well-formed log for a”} \]
\[ M \text{ soundLog } a \quad \text{“M is a sound log for a”} \]
\[ M \text{ completeLog } a \quad \text{“M is a complete log for a”} \]
\[ \neg \phi \quad \text{“not } \phi \text{”} \]
\[ \phi \land \phi' \quad \text{“} \phi \text{ and } \phi' \text{”} \]
\[ \forall m(\varphi) \quad \text{“for all } m, \varphi' \text{”} \]
\[ \Box \phi \quad \text{“henceforth } \phi \text{”} \]
\[ \Box \phi \quad \text{“so far } \phi \text{”} \]
\[ K_a(\phi) \quad \text{“a knows that } \phi \text{”} \]
\[ K_a^M (\phi) \quad \text{“if a knew M then a would know that } \phi \text{”} \]
\[ \text{CK}(\phi) \quad \text{“it is commonly known that } \phi \text{”} \]

<table>
<thead>
<tr>
<th>Table 1. Logical language</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E, i \models \text{sends}(a, M, b) ) if ( E \uplus i = B(a, M, b) )</td>
</tr>
<tr>
<td>( E, i \models \text{a receives M} ) if ( E \uplus i = R(a, M) )</td>
</tr>
<tr>
<td>( E, i \models \text{a } \alpha \text{ M} ) if ( \text{msg}(E \uplus i</td>
</tr>
<tr>
<td>( E, i \models M \text{ wfLog } a ) if ( M \in \log(a) )</td>
</tr>
<tr>
<td>( E, i \models M \text{ soundLog } a ) if ( (E, i) \models M \text{ wfLog } a ) and ( ([M]</td>
</tr>
<tr>
<td>( E, i \models M \text{ completeLog } a ) if ( (E, i) \models M \text{ wfLog } a ) and ( (E, i) \subseteq_a ([M]</td>
</tr>
<tr>
<td>( (E, i) \models \neg \phi ) if ( \text{not } (E, i) \models \phi )</td>
</tr>
<tr>
<td>( E, i \models \phi \land \phi' ) if ( (E, i) \models \phi ) and ( (E, i) \models \phi' )</td>
</tr>
<tr>
<td>( E, i \models \forall m(\varphi) ) if ( \text{for all } M \in \mathcal{M}, (E, i) \models \varphi[M/m] )</td>
</tr>
<tr>
<td>( E, i \models \Box \phi ) if ( \text{for all } j \geq i, (E, j) \models \phi )</td>
</tr>
<tr>
<td>( E, i \models \Box \phi ) if ( \text{for all } j \leq i, (E, j) \models \phi )</td>
</tr>
<tr>
<td>( E, i \models K_a(\phi) ) if ( \text{for all } (E', i') \in E^n \times N ), ( (E, i) \approx_{a} (E', i') ) then ( (E', i') \models \phi )</td>
</tr>
<tr>
<td>( E, i \models K_{a}^M (\phi) ) if ( \text{for all } M' \in \mathcal{M}, ) ( \text{if } {M'} \cap a M ) and ( (E, i) \models_{a} (E', i') ) then ( (E', i') \models \phi )</td>
</tr>
<tr>
<td>( E, i \models \text{CK}(\phi) ) if ( \text{for all } (E', i') \in E^n \times N ), ( (E, i) \approx_{a} (E', i') ) then ( (E', i') \models \phi )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2. Satisfaction relation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>For all</strong> ( a \text{ sends M to b} )</td>
</tr>
<tr>
<td><strong>For all</strong> ( a \text{ receives M} )</td>
</tr>
<tr>
<td><strong>For all</strong> ( a \text{ } \alpha \text{ M} )</td>
</tr>
<tr>
<td><strong>For all</strong> ( M \text{ wfLog } a )</td>
</tr>
<tr>
<td><strong>For all</strong> ( M \text{ soundLog } a )</td>
</tr>
<tr>
<td><strong>For all</strong> ( M \text{ completeLog } a )</td>
</tr>
<tr>
<td><strong>For not</strong> ( (E, i) \models \phi )</td>
</tr>
<tr>
<td><strong>For and</strong> ( (E, i) \models \phi ) and ( (E, i) \models \phi' )</td>
</tr>
<tr>
<td><strong>For all</strong> ( (E', i') \in E^n \times N ), ( (E, i) \approx_{a} (E', i') ) then ( (E', i') \models \phi )</td>
</tr>
<tr>
<td><strong>For all</strong> ( M' \in \mathcal{M}, ) ( \text{if } {M'} \cap a M ) and ( (E, i) \models_{a} (E', i') ) then ( (E', i') \models \phi )</td>
</tr>
<tr>
<td><strong>For all</strong> ( (E', i') \in E^n \times N ), ( (E, i) \approx_{a} (E', i') ) then ( (E', i') \models \phi )</td>
</tr>
</tbody>
</table>
- \( \preceq_a \subseteq (E^\omega \times \mathbb{N})^2 \) designates a pre-order expressing non-empty pre-fixing up to the currently inspected positions and modulo the agent’s \( a \in A \) local view, defined such that for all \((E, i), (E', i') \in E^\omega \times \mathbb{N}, \)

\[
(E, i) \preceq_a (E', i') \quad \text{iff} \quad \text{there is } E'' \in E^* \text{ such that } ((E|i)\mu a) \cdot E'' = (E'|i')\mu a \neq E''
\]

- \( \varphi[M/m] \) designates the substitution of the (closed) term \( M \) for all free occurrences of the term variable \( m \) in the formula \( \varphi \)

- \( \leq \) designates the standard total order on \( \mathbb{N} \), and \( \geq \) the converse of \( \leq \)

- \( \approx_a \subseteq (E^\omega \times \mathbb{N})^2 \) designates a relation of epistemic accessibility expressing indistinguishability of traces up to the currently inspected positions and modulo the agent’s \( a \in A \) local view, defined such that for all \((E, i), (E', i') \in E^\omega \times \mathbb{N}, \)

\[
(E, i) \approx_a (E', i') \quad \text{iff} \quad (E, i) \preceq_a (E', i') \text{ and } (E', i) \preceq_a (E, i)
\]

- \( \approx_a^* := (\bigcup_{a \in A} \approx_a)^* \), where \( * \) designates the Kleene (i.e., the reflexive transitive) closure operation

Note that the semantics of the predicate \( \text{correct} \) is application-specific, and thus left to be defined for the considered system (cf. Section B for an example). Further notice that the relation \( \preceq_a \) uniformly serves as the basis for the semantics of the epistemic accessibility relation. Finally notice that formulae \( \phi \in \Phi \) have a Herbrand-style semantics, i.e., logical constants (agent names) and functional symbols (hashing, signing, and pairing) are self-interpreted rather than interpreted in terms of (other, semantic) constants and functions. This simplifying design choice spares our framework from term-variable assignments [7]. The choice is admissible because our individuals (messages) are finite. Hence, substituting (syntactic) messages for message variables into (finite) formulae preserves the well-formedness of formulae (cf. the semantics of universal quantification).

Now, we can macro-define the following standard concepts from first-order and temporal logic: \( \top := a \wedge a \), \( \bot := \neg \top \), \( \phi \vee \phi' := \neg (\neg \phi \land \neg \phi') \), \( \phi \rightarrow \phi' := \neg \phi \lor \phi' \), \( \exists m (\phi) := \neg \forall m (\neg \phi) \), \( \forall \phi := \neg \exists \phi \) (“eventually \( \phi \)”), \( \Diamond \phi := \neg \Box \neg \phi \) (“once \( \phi \)”). Moreover, and more interestingly, we can macro-define the concepts in Table 3, which are important for our formal definition of accountability and the assumptions based on which accountability can be established. In Table 3, observe our definition of agent-based provability, which captures the idea of proofs as sufficient evidence. The definition stipulates that it be common knowledge among the agents that to the designated verifier \( a \).

1. the actual (cf. material implication) and
2. the hypothetical (cf. conditional implication)

knowledge of \( M \) be individually necessary and jointly (vacuity!) sufficient for the knowledge of \( \phi \). Note that a material conditional alone would not do here because any message unknown to \( a \) would vacuously qualify as a proof. See [18] for a detailed exposition of our provability modality.
Table 3. Important macro-definitions

<table>
<thead>
<tr>
<th>Macro-definition</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{asTrusts}(b) := K_a(\text{correct}(b))$</td>
<td>“a strongly trusts b”</td>
</tr>
<tr>
<td>faulty($a$)</td>
<td>$\neg \text{correct}(a)$</td>
</tr>
<tr>
<td>$M \vdash \phi$</td>
<td>$\exists \mathcal{R}(\exists K \left( \vdash \phi \Rightarrow K_a(\phi) \right) \land K_a(\phi))$</td>
</tr>
<tr>
<td>$M \text{ decides } \phi := (M \vdash \phi) \lor (M : \vdash \neg \phi)$</td>
<td>“M decides $\phi$ for a”</td>
</tr>
<tr>
<td>$M : \phi := \forall a(M : \vdash \phi)$</td>
<td>“M is a proof of $\phi$”</td>
</tr>
<tr>
<td>$\text{M decides } \phi := \forall a(M \text{ decides } \phi)$</td>
<td>“M decides $\phi$ for a”</td>
</tr>
<tr>
<td>$P_{(a,b)}(\phi) := \exists m((m : \vdash \phi) \land a \land m)$</td>
<td>“a can prove to b that $\phi$’s”</td>
</tr>
<tr>
<td>$P_a(\phi) := \exists m((m : \vdash \phi) \land a \land m)$</td>
<td>“a can prove that $\phi$”</td>
</tr>
<tr>
<td>$a \text{ canExpose } b := P_a(\text{faulty}(b))$</td>
<td>“M decides $\phi$ for a”</td>
</tr>
</tbody>
</table>

Finally, following [4], we define the concepts of validity, logical consequence, and logical equivalence.

**Definition 3.** A formula $\phi \in \Phi$ is valid, written $\models \phi$, if for all $(E, i) \in \mathcal{E} \times \mathbb{N}$, $(E, i) \models \phi$. A formula $\phi' \in \Phi$ is a logical consequence of a formula $\phi \in \Phi$, written $\phi \Rightarrow \phi'$, if for all $(E, i) \in \mathcal{E} \times \mathbb{N}$, if $(E, i) \models \phi$ then $(E, i) \models \phi'$. A formula $\phi' \in \Phi$ is logically equivalent to a formula $\phi \in \Phi$, written $\phi \Leftrightarrow \phi'$, if $\phi \Rightarrow \phi'$ and $\phi' \Rightarrow \phi$.

**Properties**

**Fact 1** $\models \phi \Rightarrow \phi'$ iff $\phi \Rightarrow \phi'$

**Proof.** By expansion of definitions.

The following proposition provides further useful intuitions about our framework. Technically, the proposition proves that our framework provides standard epistemic operators, and an agent-based provability operator enjoying standard properties, and that provability implies knowledge.

**Proposition 1.**

1. The auxiliary relation $\preceq$ is pre- but not partially ordering.
2. The epistemic accessibility relation $\approx_a$ is an equivalence relation. Or, equivalently, the epistemic modality $K_a$ is S5, i.e., captures the standard notion of knowledge (cf. [9] and [25, Section 7.1]):

   - $K \models K_a(\phi \Rightarrow \phi') \Rightarrow (K_a(\phi) \Rightarrow K_a(\phi'))$ (Kripke’s law / closure under implication)
   - $T \models K_a(\phi) \Rightarrow \phi$ (truth law / reflexivity)
4 \models K_a(\phi) \rightarrow K_a(K_a(\phi)) \quad (\text{positive introspection / transitivity})
5 \models \neg K_a(\phi) \rightarrow K_a(\neg K_a(\phi)) \quad (\text{negative introspection / Euclideanness})
N \text{ if } \models \phi \text{ then } \models K_a(\phi) \quad (\text{epistemic necessitation / all agents know all validities})

3. Being defined in terms of an equivalence relation, CK is S5 too. In particular, if \models \phi \text{ then } \models CK(\phi), \text{ i.e., validities are common knowledge. Further, } 
\models CK(\phi) \rightarrow \forall a(CK_a(\phi)) \text{ and } \models CK(\phi \rightarrow \forall a(CK_a(\phi))) \rightarrow (\phi \rightarrow CK(\phi)) \ [25, \text{ Section 7.1}].

4. \models S_4, \text{ i.e., captures a standard (though even interactive) notion of Gödel-style provability [2]: } 
\begin{align*}
K & \models P_a(\phi \rightarrow \phi') \rightarrow (P_a(\phi) \rightarrow P_a(\phi')) \\
T & \models P_a(\phi) \rightarrow \phi \\
4 & \models P_a(\phi) \rightarrow P_a(P_a(\phi)) \\
N & \text{ if } \models \phi \text{ then } \models P_a(\phi) \\
5 & \models P_a(\phi) \rightarrow K_a(\phi).
\end{align*}

Proof. For the failure of the anti-symmetry of the relation \preceq_a consider the following counter-example with \(a, b \in A\) such that \(a \neq b\), \(M \in M\), and \(E \in E^a\): 
\((S(a, M, b) \cdot E, 1) \approx_a (S(a, M, b) \cdot R(b, M) \cdot E, 2)\), but \((S(a, M, b) \cdot E, 1) \neq (S(a, M, b) \cdot R(b, M) \cdot E, 2)\). Thus the culprits for the failure are trace positions and the (necessary) projection of the trace onto agent (local) views.

The equivalence property of \approx_a follows from the pre-order property of \preceq_a.

For the proof of the properties of the provability modality and the law connecting provability to knowledge, see [18].

Lemma 1. Individual knowledge is never forgotten. Formally, 
\[ \models \forall a \forall m (\square(a \land m \rightarrow \square(a \land m))). \]

Proof. Straightforward from definitions.

This lemma obviously depends on persistent storage for the essential parts of an agent’s individual knowledge (i.e., those parts that cannot be reconstructed from other parts).

The following nice-to-know proposition gives guarded Barcan laws (cf. first-order epistemic logic [7]), i.e., guarded quantifiers (w.r.t. to individual knowledge of say \(a\)) can be freely extruded from and intruded into the scope of epistemic modalities (capturing propositional knowledge w.r.t. \(a\)).

Proposition 2 (Guarded Barcan laws).
\begin{enumerate}
\item \models K_a(\exists m (a \land m \land \phi)) \leftrightarrow \exists m (a \land m \land K_a(\phi))
\item \models K_a(\forall m (a \land m \rightarrow \phi)) \leftrightarrow \forall m (a \land m \rightarrow K_a(\phi))
\end{enumerate}

Proof. Straightforward from definitions.
2.2 Pre-establishment

An important particularity of our framework is that it is parametric in the proof of a few key lemmata, whose contents we state as assumptions, and to which we reduce the soundness and the completeness proof of potential applications a priori, i.e., before their design. That is, our framework pre-establishes the soundness and the completeness aspect of accountability by factoring the proof of each aspect into the contingent (i.e., application-specific) and the logical (i.e., conceptual) content. As a result, the accountability proof of a particular system is reduced to the proof of the key lemmata (i.e., to the discharge of the key assumptions), which the system designer needs to establish for a considered system a posteriori, i.e., after its design.

More precisely, the factoring confines the contingent content to a finite conjunction \( \phi' \in \Phi \) of system assumptions that logically implies the considered system goal \( \phi \in \Phi \) (i.e., soundness or completeness), which contains the logical content, i.e., \( \phi' \Rightarrow \phi \). Then, given a definition of correct for the considered system and a finite conjunction \( \phi'' \) of system axioms\(^6\) that logically implies the system assumptions \( \phi' \), i.e., \( \phi'' \Rightarrow \phi' \), the system axioms logically imply the system goal, i.e., \( \phi'' \Rightarrow \phi \). Note that a definition of correct may entail a system assumption or axiom to become a system validity, and thus common knowledge, as already indicated in Proposition 1.3. This is a useful fact for formal proofs, which we will develop in Fitch-style natural deduction. (Recall that Fitch-style proofs are read outside-in.)

**Key assumptions** We make three succinct key assumptions that are sufficient for achieving the soundness and the completeness aspect of accountability in distributed systems. Our assumptions are also necessary for faithful modelling, i.e., not making one of these assumptions would imply not modelling faithfully these systems. In particular, faithful modelling requires the use of logs (i.e., accounting entries) of some form (cf. Section B for an example).

Two of our assumptions cannot be discharged by proof but only by psychology and physics because their discharge depends on the system user and/or the communication medium. The other assumption can be discharged by proof because its discharge only depends on the mathematics that models the cryptographic mechanisms (e.g., hashing and signing) required for the system implementations. However, observe that our assumptions are fully abstract w.r.t. these mechanisms in the sense that the assumptions do not mention functional symbols, which represent these mechanisms at the term-language level.

**Assumption 1** Decisive logs are necessarily known (though not necessarily filed, e.g., by corrupt agents)\(^7\). Formally,

\[
\text{\(A1 := \forall a \exists m(a, m \land \text{decisiveLog} \ a)\).}
\]

---

\(^6\) Here, a system axiom is a sentence stipulating a characteristic property of the considered system, and not necessarily an axiom in a proof system.

\(^7\) Recall from the last three lines of Table 3 that, as opposed to being a filed log, being a mere log is a mere well-formedness criterion.
This assumption can be discharged by proof, on the condition that correct (being part of decisiveLog, cf. Table 3) be defined for the considered system.

**Assumption 2** Faultiness is persistently provable by filed logs. Formally,

\[ A2 := \forall a \Box (\text{faulty}(a) \rightarrow \exists m (m \text{ filedLog} a \land \Box (m : \text{faulty}(a)))) \]

This assumption cannot be discharged by proof due to its dependence on system users (e.g., the willingness of \( a \) to file a log \( m \)). Observe that this assumption implies the assumption (made also by the PeerReview and other systems [13, Section 4.4]) that log files cannot be tampered with.

**Assumption 3** Logs are eventually known. Formally,

\[ A3 := \forall a \forall m (a \text{ logs } m \rightarrow \Box \forall b (b 	ext{ k m})) \]

This assumption (made also by the PeerReview and other systems [13, Section 4.3]) cannot be discharged by proof either, due to its dependence on, again, system users (e.g., the willingness of \( a \) to commit logs \( m \)) and also, the communication medium (e.g., the reliability of the communication channels).

**Soundness theorem** As mentioned before, soundness in accountability means that correct agents can prove their correctness to the other agents.

We can transcribe this natural-language formulation into our formal language with the following macro-definition:

\[ \text{Soundness} := \forall a \Box (\text{correct}(a) \rightarrow P_a (\text{correct}(a))) \]

Observe that soundness builds trust in the sense of Table 3, Line 1, i.e., the proof of an agent \( a \)’s correctness to another agent, say \( b \), induces the knowledge with \( b \) that \( a \) is correct (cf. [20] for a detailed exposition of this sense of trust, with example applications in cryptographic-key management). Also notice that in Soundness, the converse implication is for free due to the truth law of \( P_a \) for arbitrary \( a \in A \) (cf. Proposition 1.4.T).

**Theorem 1** (Accountability soundness).

1. **Assumption A1 is a sufficient condition for Soundness.** Formally,

\[ A1 \Rightarrow \text{Soundness} \]

2. **Whenever accountability is log-based (which is almost always the case in practice), Assumption A1 also is a necessary condition for Soundness.**

Proof. For (1), see Table 4. For (2), consider that if there is an agent such that eventually all her known logs are non-decisive (this is the negation of A1) then she has no (log-based) means of proving her correctness — even when she does happen to be correct.
Completeness theorem As mentioned before, completeness in accountability means that all agents can eventually always prove the faultiness of faulty agents.

We can transcribe this natural-language formulation into our formal language with the following macro-definition:

\[
\text{Completeness} := \forall a \exists b (\text{faulty}(a) \rightarrow \forall b \exists c (\text{canExpose} a)).
\]

Theorem 2 (Accountability completeness). Assumptions A2 and A3 jointly are a sufficient condition for Completeness. Formally,

\[A2 \land A3 \Rightarrow \text{Completeness}.\]

Proof. See Table 5.

Accountability theorem As mentioned before, accountability is the conjunction of accountability soundness and accountability completeness.

We macro-define,

\[
\text{Accountability} := \text{Soundness} \land \text{Completeness}.
\]

Theorem 3 (Accountability).

1. Assumptions A1, A2, and A3 jointly are a sufficient condition for Completeness. Formally,

\[A1 \land A2 \land A3 \Rightarrow \text{Accountability}\]

2. Whenever accountability is log-based, Assumption A1 also is a necessary condition for Accountability.

Proof. By Theorem 1 and Theorem 2.

3 Conclusion

3.1 Assessment

Our goal has been to take up and meet [28]'s challenge of developing a general and practical methodology and technique that can approach the ideal of full accountability for networked systems.

We have delivered this methodology and technique within a multi-modal, model-theoretic framework that is parametric in an application-specific

1. set \( \mathcal{L} \) of well-formed logs (cf. Definition 1)
2. predicate correct for agent correctness (cf. Table 1).
The methodology consists in proving that the characteristic properties of a considered system (i.e., the system axioms) logically imply three logically sufficient and modelling-wise necessary conditions for accountability.

The technique then consists in:

1. instantiating $L$ and correct for the considered system
2. formalising the system axioms with the powerful descriptive abstractions built-in in our framework, i.e., the modalities, most notably Gödel-style provability
3. proving the implication by exploiting the inferential content of these modalities.

Finally, we have illustrated the applicability of our framework on the case study of the Distributed Book-Keeping (DBK) design pattern, which we introduced as an analogue for distributed systems of Pacioli’s famous double-entry book-keeping principle (cf. Section B).

We have defined DBK in terms of a distributed accounting daemon, which:

1. pinpoints an instance of Russel’s paradox for distributed systems
2. provides falsifiability of agent correctness in the sense of Popper
3. effects a timeline entanglement in the sense of [24].

On top of the accounting daemon, a distributed auditing daemon can provide also verifiability of agent correctness by encouraging agents to produce proof of their behavioural status (correct or faulty). Such proofs could be encouraged by creating the appropriate deterrents and incentives for the agents in the considered system.

### 3.2 Related work

The following three formal approaches to accountability with a general aim exist.

To the best of our knowledge, [17] is the first to have published the idea of formalising accountability with a notion of provability. However, the author seems to have been unaware of standard modal provability, as corresponding work is not mentioned in his paper. Rather, the author construes a supposed notion of provability from two postulates: if an agent can prove two statements then the agent can prove the conjunction of them, and if an agent can prove a statement and that statement implies another statement then the agent can prove the implied statement. Hence, the author’s notion of provability need not even respect the truth law, i.e., that the provability of a statement imply the truth of that statement.

To the best of our knowledge, [3] are the first to have published the idea of proof as sufficient evidence. However, the authors do not seem to have been aware of standard modal provability either, nor do they have a general account of provability in the sense that they “can only formalize [protocol, not agent] correctness with respect to a specific protocol”, which is a proof (in our sense) that their logical formalisation of provability is not satisfactory yet.
The same comment applies for the same reason to [21], where accountability-related properties for electronic contract-signing are studied in terms of Alternating Temporal Logic (ATL).

Then, in [15], a process-algebraic programming model for distributed systems for accountability and audit is presented in the particular context of authorisation. In that approach, finitary systems are compiled to turn-based games, and ATL is used to specify system properties. The authors actually call their property formulations “logical encodings”. In contrast, we define the properties in our logic directly as mere transcriptions of their natural-language formulations. Hence again, a similar comment as for the previous approach can be made.

Finally, in [11], a type-based definition of log-based auditability is given. Thereby, the authors informally refer to concepts like provability (w.r.t. identity, authenticity), agreement (w.r.t. public-key infrastructures, properties of judges), knowledge (w.r.t. properties of judges), and trust (w.r.t. agents, functions, function libraries). All these concepts can be captured formally in the logical language of our framework: provability with the provability modality, agreement with the common-knowledge modality, (property) knowledge with the knowledge modality, and trust as the knowledge of (the property of) agent correctness.

Future work

In future work, we would like to employ our framework in a provability-based study of the Accountable Internet Protocol [1]. Given that our notion of provability is defined in terms of knowledge, it could be possible to reduce such a provability-based study to the knowledge-based study of the Internet Protocol without accountability [27].

References

## A Formal proofs

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$(E, i) \in \mathcal{E} \times \mathbb{N}$ hyp.</td>
</tr>
<tr>
<td>2.</td>
<td>$(E, i) \models A_1$ hyp.</td>
</tr>
<tr>
<td>3.</td>
<td>$a \in A$ hyp.</td>
</tr>
<tr>
<td>4.</td>
<td>$j \geq i$ hyp.</td>
</tr>
<tr>
<td>5.</td>
<td>$(E, j) \models \text{correct}(a)$ hyp.</td>
</tr>
<tr>
<td>6.</td>
<td>there is $M \in \mathcal{M}$ s.t. $(E, j) \models a \land M$ and $\text{decisiveLog}(M)$ hyp.</td>
</tr>
<tr>
<td>7.</td>
<td>$M \in \mathcal{M}$ and $(E, j) \models a \land M$ and $\text{decisiveLog}(M)$ hyp.</td>
</tr>
<tr>
<td>8.</td>
<td>$(E, j) \models M : \text{correct}(a) \lor M : \neg \text{correct}(a)$ hyp.</td>
</tr>
<tr>
<td>9.</td>
<td>$(E, j) \models \neg (M : \neg \text{correct}(a))$ hyp.</td>
</tr>
<tr>
<td>10.</td>
<td>$(E, j) \models M : \text{correct}(a)$ hyp.</td>
</tr>
<tr>
<td>11.</td>
<td>$(E, j) \models a \land M$ hyp.</td>
</tr>
<tr>
<td>12.</td>
<td>$(E, j) \models \text{P}_a(\text{correct}(a))$ hyp.</td>
</tr>
<tr>
<td>13.</td>
<td>$(E, j) \models \text{P}_a(\text{correct}(a))$ hyp.</td>
</tr>
<tr>
<td>14.</td>
<td>$(E, j) \models \text{correct}(a) \rightarrow \text{P}_a(\text{correct}(a))$ hyp.</td>
</tr>
<tr>
<td>15.</td>
<td>$(E, i) \models \text{correct}(a) \rightarrow \text{P}_a(\text{correct}(a))$ hyp.</td>
</tr>
<tr>
<td>16.</td>
<td>$(E, i) \models \forall \Box \text{correct}(a) \rightarrow \text{P}_a(\text{correct}(a))$ hyp.</td>
</tr>
<tr>
<td>17.</td>
<td>$(E, i) \models \text{Soundness}$ hyp.</td>
</tr>
<tr>
<td>18.</td>
<td>$(E, i) \models A_1 \rightarrow \text{Soundness}$ hyp.</td>
</tr>
<tr>
<td>19.</td>
<td>$A_1 \Rightarrow \text{Soundness}$ hyp.</td>
</tr>
</tbody>
</table>
### Table 5. Accountability completeness

<table>
<thead>
<tr>
<th>Step</th>
<th>Condition</th>
<th>Hypothesis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>$(E, i) \in \mathcal{E}^* \times \mathbb{N}$</td>
<td>hyp.</td>
</tr>
<tr>
<td>2.</td>
<td>$(E, i) \models A_2 \land A_3$</td>
<td>hyp.</td>
</tr>
<tr>
<td>3.</td>
<td>$a \in A$</td>
<td>hyp.</td>
</tr>
<tr>
<td>4.</td>
<td>$j \geq i$</td>
<td>hyp.</td>
</tr>
<tr>
<td>5.</td>
<td>$(E, j) \models \text{faulty}(a)$</td>
<td>hyp.</td>
</tr>
<tr>
<td>6.</td>
<td>$b \in A$</td>
<td>hyp.</td>
</tr>
<tr>
<td>7.</td>
<td>there is $M \in M$ s.t.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$(E, j) \models M \text{filedLog } a \land \Box(M : \text{faulty}(a))$</td>
<td>2[A2], 3, 4, 5</td>
</tr>
<tr>
<td>8.</td>
<td>$M \in M$ and $(E, j) \models M \text{filedLog } a \land \Box(M : \text{faulty}(a))$</td>
<td>hyp.</td>
</tr>
<tr>
<td>9.</td>
<td>$(E, j) \models M \text{filedLog } a$</td>
<td>8</td>
</tr>
<tr>
<td>10.</td>
<td>$(E, j) \models \Box(b \land b \land M)$</td>
<td>2[A3], 6, 9</td>
</tr>
<tr>
<td>11.</td>
<td>$(E, j) \models \Box(M : \text{faulty}(a))$</td>
<td>6, 10, Lemma 1</td>
</tr>
<tr>
<td>12.</td>
<td>$(E, j) \models \Diamond(M : \text{faulty}(a))$</td>
<td>8</td>
</tr>
<tr>
<td>13.</td>
<td>$(E, j) \models \Diamond(M : \text{faulty}(a))$</td>
<td>12</td>
</tr>
<tr>
<td>14.</td>
<td>$(E, j) \models \Diamond((M : \text{faulty}(a)) \land b \land b \land M)$</td>
<td>11, 13</td>
</tr>
<tr>
<td>15.</td>
<td>$(E, j) \models \Box\Box((M : \text{faulty}(a)) \land b \land b \land m)$</td>
<td>14</td>
</tr>
<tr>
<td>16.</td>
<td>$(E, j) \models \Box\Diamond_p(M : \text{faulty}(a))$</td>
<td>15</td>
</tr>
<tr>
<td>17.</td>
<td>$(E, j) \models \Box\Diamond(b \text{canExpose } a)$</td>
<td>16</td>
</tr>
<tr>
<td>18.</td>
<td>$(E, j) \models \Box\Diamond(b \text{canExpose } a)$</td>
<td>7, 8–17</td>
</tr>
<tr>
<td>19.</td>
<td>$(E, j) \models \forall b\Diamond(b \text{canExpose } a)$</td>
<td>6–18</td>
</tr>
<tr>
<td>20.</td>
<td>$(E, j) \models \text{faulty}(a) \rightarrow \forall b\Diamond(b \text{canExpose } a)$</td>
<td>5–19</td>
</tr>
<tr>
<td>21.</td>
<td>$(E, i) \models \Diamond(\text{faulty}(a) \rightarrow \forall b\Diamond(b \text{canExpose } a))$</td>
<td>4–20</td>
</tr>
<tr>
<td>22.</td>
<td>$(E, i) \models \forall a(\text{faulty}(a) \rightarrow \forall b\Diamond(b \text{canExpose } a))$</td>
<td>3–21</td>
</tr>
<tr>
<td>23.</td>
<td>$(E, i) \models \text{Completeness}$</td>
<td>22</td>
</tr>
<tr>
<td>24.</td>
<td>$(E, i) \models (A_2 \land A_3) \rightarrow \text{Completeness}$</td>
<td>2–23</td>
</tr>
<tr>
<td>25.</td>
<td>$A_2 \land A_3 \Rightarrow \text{Completeness}$</td>
<td>1–24</td>
</tr>
</tbody>
</table>
B Case study

In this section, we illustrate the applicability of our framework on the principled, real-world case study of the Distributed Book-Keeping (DBK) design pattern. Our case study is real-world in the sense that the intuition of the design pattern is already implicit in the design of Byzantine Fault Detection [12] and in its implementation, the PeerReview system [13]. The case study is meant to exemplify how to distil the declarative essence of an accountable distributed system in our multi-modal, model-theoretic framework.

B.1 The Distributed Book-Keeping design pattern DBK

Distributed Book-Keeping (DBK) is a design pattern for accountable distributed systems, which we define hereafter in terms of a distributed accounting daemon.

DBK is the analogue for distributed systems of Fra Luca Bartolomeo de Pacioli’s (1446/7–1517) codification of the Venetian double-entry book-keeping principle for financial transactions. In DBK, the analogue of a financial transaction is a message-passing communication, and the analogue of a crediting or debiting account entry is a sending- or receiving-event log entry, respectively. More precisely in DBK, the sending or receiving of a data message by an agent must match the immediate sending of an adequate log message of the corresponding quoted data-message event to the other agents. (Bear in mind that we cannot transmit events as such, i.e., without quotation. After all, a sending or reception event symbolises the very transmission of a message. Thus we do need to marshal/serialise events for their own transmission, which is what their quotation symbolises.) DBK is a design pattern in the sense that DBK delegates the specification of the following control and data refinements to designs:

– resolution of non-determinism (e.g., scheduling of broadcasts)
– utilisation of storage (e.g., choice of data structures)
– computational efficiency (e.g., hashing).

The designs in turn delegate the realisation of these specifications to implementations.

However, what DBK does specify is the definition of the predicate correct, and it does so by means of the distributed accounting daemon shown in Table 6, as follows:

\[(E, i) \models \text{correct}(a) \iff (E[i], a) \text{ is compatible with } \text{Accounting-Daemon}(a),\]

where to be compatible with agent a’s accounting daemon means for the finite trace \((E[i], a)\) to exhibit the event \(S(a, L', c)\) (generated by the action \(\text{send}(L', c)\)) whenever a WHEN-guard applies at its currently inspected position — for all positions (cf. Appendix B.3 for a formal definition in terms of a quadratic-time functional program).

\text{Accounting-Daemon} employs the following notable programming constructs:
Table 6. Distributed accounting daemon

<table>
<thead>
<tr>
<th>PROGRAM Accounting-Daemon (a : agent) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP SUCH-THERE (b : agent) AND (D : data) {</td>
</tr>
<tr>
<td>WHEN sends(a, sign(D, a), b) {</td>
</tr>
<tr>
<td>FOR EVERY agent c EXCEPT (a,b) {</td>
</tr>
<tr>
<td>CHOOSE log L SUCH-THERE adequateLog(L', a) {</td>
</tr>
<tr>
<td>send(L', c)</td>
</tr>
<tr>
<td>} WHERE L' = sign((quote(S(a, sign(D, a), b)), L), a)</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>WHEN receives(a, sign(D, b)) {</td>
</tr>
<tr>
<td>FOR EVERY agent c EXCEPT a {</td>
</tr>
<tr>
<td>CHOOSE log L SUCH-THERE adequateLog(L', a) {</td>
</tr>
<tr>
<td>send(L', c)</td>
</tr>
<tr>
<td>} WHERE L' = sign((quote(R(a, sign(D, b))), L), a)</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

- sends(a, sign(D, a), b) and receives(a, sign(D, b)), in logical notation sends(a, [D]a, b) and a receives [D]b, respectively
- the declarative programming abstraction CHOOSE—SUCH-THERE for storage utilisation, which we use as an analogue in our programming setting of Hilbert’s choice operator in logical settings (cf. [6], where this operator is also mentioned as a specification construct in Abstract-State-Machine programs), e.g., “CHOOSE x SUCH-THERE P(x)” chooses an x such that x has the property P
- adequateLog(L', a), in logical notation L' adequateLog a, whose truth condition relies on the two parameters of our framework, namely:
  1. the set (and function) L of (an agent’s) well-formed logs, which for all ∈ and a ∈ A we fix as

\[ L \ni L := \{(\varepsilon) \in L : (\varepsilon, L) \}

2. the obvious function [·] : L → E* transcribing log messages to event traces.

Observe that inside the accounting daemon, the sending of a log that is complete a priori (i.e., before sending) immediately entails its incompleteness a posteriori (i.e., after sending). In other words, sending a log that is also complete a posteriori is impossible due to Russel’s paradox — the log would have to contain itself. (This state of affairs will motivate the introduction of semi-decisive logs in the next section.) Hence without further ado, we have the following two validities, and thus pieces of common knowledge among agents (cf. Proposition 1.3).
Fact 2 (Intrinsic incompleteness of committed logs) For all agents $a \neq b$ and messages $M$:

1. $\models a \text{ logs } M \rightarrow \neg (M \text{ completeLog } a)$
2. $\models (b \land M \land M \text{ wfLog } a) \rightarrow \neg (M \text{ completeLog } a)$

Whence the following two corollaries with practical impacts on the programming of accountable distributed systems and the philosophy thereof.

**Corollary 1 (The purpose of audit).** An agent’s correctness can only be proven by that very agent itself outside the accounting daemon, e.g., within the protocol of a meta- or auditing daemon, which is why the addition of an auditing daemon is desirable.

**Corollary 2 (Falsifiability of agent correctness).** The other agents must content themselves with the possibility of proving an agent’s faultiness if so, i.e., the accounting daemon exactly provides falsifiability in the sense of Popper’s critical rationalism, namely Popper’s dictum that a hypothesis (here, agent correctness) should be falsifiable in the sense that if the hypothesis is false then its falsehood should be cognisable (here, by another agent).

Further observe that Accounting-Daemon only logs sent and received data but not log messages because logging log messages would entail an explosion in network traffic, and thus immediately saturate the network bandwidth. However, it is conceivable to refine Accounting-Daemon with bounded logging of log messages, which would induce logging degrees, which in turn would induce degrees of mutuality of knowledge between agents. Recall that mutuality of knowledge between agents can be expressed with nested knowledge modalities (e.g., $K_a(K_b(K_a(\phi))))$, but that the evaluation of the truth of the resulting formulae requires the communication of at least one acknowledgement per nesting between the agents. Ultimately, logging degrees induce degrees of accountability since accountability is defined in terms of provability, which in turn is defined in terms of knowledge (cf. Table 2).

Finally note that Accounting-Daemon actually effects a timeline entanglement in the sense of [24], such that the effectuated entanglement is:

- all-to-all, i.e., all agents are to entangle their timelines with all other agents
- data event-driven, i.e., only data sending and reception events are to be entangled
- all-inclusive, i.e., all other agents’ logs are to be included in an agent’s own logs.

For details on timeline entanglement, see [24].

**B.2 Results**

DBK yields the discharge of Assumption A1 as Corollary 4 by way of Corollary 3 and Lemma 2.
Lemma 2. $\forall a \exists m (a \land m \text{adequateLog } a)$

Proof. By inspection of the definition of $k$ and $\text{adequateLog}$.

Corollary 3. $\forall m \forall a (m \text{adequateLog } a \rightarrow m \text{decisiveLog } a)$

Proof sketch 1. By induction over the structure of logs $L \in L$ employing disjunction of the contrary cases for $a \in A$ to be or not to be correct.

The base case is $\models ["\epsilon(a)"]_a \text{adequateLog } a \rightarrow ["\epsilon(a)"]_a \text{decisiveLog } a$.

and the inductive case is $\models (["\epsilon(a)",L])_a \text{adequateLog } a \land L \text{decisiveLog } a \rightarrow ["\epsilon(a)",L])_a \text{decisiveLog } a$.

Corollary 4. $\models A1$

Proof. By Lemma 2 and Corollary 3.

Hence,

$\models \text{Soundness}.$

That is, our distributed accounting daemon ensures soundness. (Dually, a distributed auditing daemon encourages completeness.)

Finally, DBK yields the conditional discharge of Assumption A2 as Corollary 6 by way of Lemma 4 and Corollary 5 from Lemma 3. (Recall from Section 2.2 that the assumptions for accountability completeness cannot be fully discharged by proof.)

Lemma 3. $\models \forall a(\text{correct}(a) \rightarrow \forall m (m \text{filedLog } a \rightarrow m \text{soundLog } a))$

Proof. By inspection of the definition of $\text{correct}$.

Corollary 5.

$\models \forall m \forall a (m \text{filedLog } a \rightarrow m \text{semiDecisiveLog } a)$

where $m \text{semiDecisiveLog } a := \neg (m \text{soundLog } a) \rightarrow m : \text{faulty}(a)$.

Proof. See Table 7, where: (1) Line 8 follows from Lemma 3 and the necessitation law (N) for $K_b$ (cf. Proposition 1). (2) Line 12 follows from Line 5 and 6, and the facts that (2.1) logs (e.g., $M$) bear the signature of their generator (here $a$, thus $(E,j) \models K_b(M \text{filedLog } a)$), and (2.2) unsound logs (here $M$) in a given system state (here $(E,j)$) of the system execution tree remain unsound in any other (in particular $K_b$-accessible) state of the same tree (thus $(E,j) \models K_b(\neg(M \text{soundLog } a))$). (3) Line 13 and 14 follow both from Line 11 and 12 by Kripke’s law (K) for $K_b$ (cf. Proposition 1). (4) Common knowledge in Line 15 follows from the fact that the form and effect of logs is (pre-established) common knowledge.
Table 7. The accounting daemon forces semi-decisive logs

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>((E, i) \in \mathcal{E}^i \times \mathbb{N})</td>
</tr>
<tr>
<td>2.</td>
<td>(M \in \mathcal{M})</td>
</tr>
<tr>
<td>3.</td>
<td>(a \in \mathcal{A})</td>
</tr>
<tr>
<td>4.</td>
<td>(j \geq i)</td>
</tr>
<tr>
<td>5.</td>
<td>((E, j) \models M \text{ filedLog } a)</td>
</tr>
<tr>
<td>6.</td>
<td>\text{not} ((E, j) \models M \text{ soundLog } a)</td>
</tr>
<tr>
<td>7.</td>
<td>(b \in \mathcal{A})</td>
</tr>
<tr>
<td>8.</td>
<td>(\models K_b(\forall a(\text{correct}(a) \rightarrow \forall m(m \text{ filedLog } a \rightarrow m \text{ soundLog } a))))</td>
</tr>
<tr>
<td>9.</td>
<td>(\models K_b(\forall a(\text{correct}(a) \rightarrow (m \text{ filedLog } a \rightarrow m \text{ soundLog } a))))</td>
</tr>
<tr>
<td>10.</td>
<td>(\models K_b(\text{correct}(a) \rightarrow (M \text{ filedLog } a \rightarrow M \text{ soundLog } a)))</td>
</tr>
<tr>
<td>11.</td>
<td>(\models K_b((M \text{ filedLog } a \land \neg(M \text{ soundLog } a)) \rightarrow \text{faulty}(a)))</td>
</tr>
<tr>
<td>12.</td>
<td>((E, j) \models b \land M \rightarrow K_b(\text{faulty}(a)))</td>
</tr>
<tr>
<td>13.</td>
<td>((E, j) \models b \land M \rightarrow K_b(\text{faulty}(a)))</td>
</tr>
<tr>
<td>14.</td>
<td>((E, j) \models K_b^M(\text{faulty}(a)))</td>
</tr>
<tr>
<td>15.</td>
<td>((E, j) \models \text{CK}((b \land M \rightarrow K_b(\text{faulty}(a))) \land K_b^M(\text{faulty}(a))))</td>
</tr>
<tr>
<td>16.</td>
<td>((E, j) \models M \rightarrow \text{faulty}(a))</td>
</tr>
<tr>
<td>17.</td>
<td>((E, j) \models M \text{ semiDecisiveLog } a)</td>
</tr>
<tr>
<td>18.</td>
<td>((E, j) \models M \text{ semiDecisiveLog } a)</td>
</tr>
<tr>
<td>19.</td>
<td>((E, j) \models \Box(M \text{ semiDecisiveLog } a))</td>
</tr>
<tr>
<td>20.</td>
<td>((E, j) \models \forall a(\Box(M \text{ semiDecisiveLog } a) \rightarrow m \text{ semiDecisiveLog } a))</td>
</tr>
<tr>
<td>21.</td>
<td>((E, j) \models \forall m(\forall a(\Box(m \text{ filedLog } a) \rightarrow m \text{ semiDecisiveLog } a)))</td>
</tr>
<tr>
<td>22.</td>
<td>(\models \forall m(\forall a(\Box(m \text{ filedLog } a) \rightarrow m \text{ semiDecisiveLog } a)))</td>
</tr>
</tbody>
</table>

Lemma 4. \(\models \forall a(\Box(\text{faulty}(a) \rightarrow \Box(\text{faulty}(a)))\)

Proof. By inspection of the definition of correct.

Corollary 6. If \(\models \forall a(\Box(\text{faulty}(a) \rightarrow \exists m(m \text{ filedLog } a)))\) then \(\models A2\).

Proof. By Corollary 5 and Lemma 4.

Hence,

If \(\models \forall a(\Box(\text{faulty}(a) \rightarrow \exists m(m \text{ filedLog } a)))\)

then A3 \(\Rightarrow\) Completeness.

Notice the three different implications.

B.3 Formal definition of agent correctness in DBK

We present a definition of \(\text{correct}\) in Table 8. For concreteness, our presenta-
Table 8. Functional program for the correctness predicate.

```ocaml
let rec prefix xs ys =  
  match xs, ys with  
  | x::xs', y::ys' when x = y ->  
    prefix xs' ys'  
  | [] -> true  
  | _ -> false

let msgOf e =  
  match e with  
  | S (_, m, _) | R (_, m) -> m  
  | _ -> false

let sender e =  
  match e with  
  | S (a, _, _) | R (a, _) -> a  
  | _ -> false

let rec containsQuoted m =  
  match m with  
  | BaseData _ | Agent _ -> false  
  | Hash m' | Signed (m', _) ->  
    containsQuoted m'  
  | Pair (m', m") ->  
    containsQuoted m' || containsQuoted m"  
  | _ -> true

let proj es a =  
  List.filter (fun e -> sender e = a) es

let rec logToTrace l =  
  match l with  
  | Signed (Quoted (e), a) -> [e]  
  | Signed (Pair (Quoted (e), l'), a) ->  
    (logToTrace l') @ [e]  
  | _ -> raiseLogNotFound

let soundLog a l es =  
  prefix (proj l a) (proj es a)  
  let completeLog a l es =  
    prefix (proj es a) (proj l a)  
  let adequateLog a l es =  
    soundLog a l es && completeLog a l es  
  let rec findLogs a agents es =  
    if is_empty agents then [], es  
    else  
    match es with  
    | R (c, Signed (_, m)) | l as l' when  
      mem c agents && a' = a && wfLog l ->  
      let ls, es" =  
        findLogs a (remove c agents) es' in  
      l::ls, es"  
    | _ -> raiseLogNotFound

let rec correctIter a agents sofar rest =  
  match rest with  
  | e when sender e = a then  
    let sofar' = sofar @ [e] in  
    let ls, rest" =  
      findLogs a (remove c agents) sofar' in  
    List.for_all  
      (fun l ->  
        adequateLog  
        (logToTrace l) sofar')  
      ls  
    &&  
    correctIter a agents sofar' rest"  
  | _ -> true

let correct a agents es =  
  try correctIter a agents [] es  
  with LogNotFound -> false
```
Table 9. Data types for the correctness predicate.

<table>
<thead>
<tr>
<th>Line</th>
<th>OCaml Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>type basedata = string</td>
</tr>
<tr>
<td>2</td>
<td>type agent = string</td>
</tr>
<tr>
<td>3</td>
<td>type message =</td>
</tr>
<tr>
<td>4</td>
<td>BaseData of basedata</td>
</tr>
<tr>
<td>5</td>
<td>Agent of agent</td>
</tr>
<tr>
<td>6</td>
<td>Hash of message</td>
</tr>
<tr>
<td>7</td>
<td>Signed of message * agent</td>
</tr>
<tr>
<td>8</td>
<td>Pair of message * message</td>
</tr>
<tr>
<td>9</td>
<td>Quoted of event</td>
</tr>
<tr>
<td>10</td>
<td>and event =</td>
</tr>
<tr>
<td>11</td>
<td>S of agent * message * agent</td>
</tr>
<tr>
<td>12</td>
<td>R of agent * message</td>
</tr>
<tr>
<td>13</td>
<td>exception LogNotFound</td>
</tr>
</tbody>
</table>

The function uses a functional style and is given in the functional programming language OCaml. The function defining correct is shown on Line 68. We describe its components below.

First, Table 9 declares OCaml data types that represent messages and events according to Definition 2. We assume that base-data items and agent names are strings. Messages and events are terms of respective algebraic data types, which are mutually recursive. We represent (finite) traces by lists of events, where the head of the list is the first event in the trace. We will use exceptions when checking logs, hence a corresponding exception declaration. Next, we define auxiliary functions. The function `msgOf` extracts the message from a given event using pattern-matching. `sender` returns the sender of a sending event. `containsQuoted` tests if a message contains a quoted event `Quoted(...)`. `proj` projects a trace onto an agent’s view using a standard library function that filters out non-local events, cf. ↓ on Page 7. We check if a message represents a well-formed log using `wfLog`, as defined on Page 20. Such logs can be transcribed into events by applying the function `logToTrace`, while any attempt to transcribe a message that is not a well-formed log raises an exception.

The compatibility condition requires that each send and receive event is adequately logged, and the log distribution is required to take place immediately after the event occurs. The functions `soundLog`, `completeLog`, and their composition `adequateLog` are used to compare a logged sequence of events with a given trace, cf. `soundLog`, `completeLog`, and `adequateLog` in Table 3. They use a standard function `prefix` that tests the prefix relation on pairs of lists, which is applied on projected traces. We find logging events using the function `findLogs`. It succeeds if a trace `es` contains a log event for each agent that is participating in the system and is required to receive a log according to the accounting daemon.

We assume that agent names are kept in a set `agents`, which we manipulate using standard functions `is_empty`, `mem`, and `remove` that check for emptiness, element membership, and remove an element, respectively. To find a log event for each participating agent, `findLogs` scans the trace until it finds a log event for each participating agent. As a result, `findLogs` either returns a list of log events or an exception.
Table 10. Functional program for the accounting daemon.

```ocaml
let adeqLog : message ref = ref (BaseData "")

let logSend a m b =
  adeqLog := Signed (Pair (Quoted (S (a, m, b)), !adeqLog), a)

let logRecv a m =
  adeqLog := Signed (Pair (Quoted (R (a, m)), !adeqLog), a)

let send m b = ()

let mkLogMsg m = Pair (m, BaseData "log")

let isLogMsg m =
  match m with |
  | Pair(_, BaseData "log") -> true |
  | _ -> false

let onSend a m b agents =
  match m with |
  | Signed (m', _) when a = a' && not(isLogMsg m') ->
  | logSend a m b ;
  | List.iter (fun c ->
    if c <> a && c <> b then
    send (mkLogMsg !adeqLog) c)
  ) agents |
  | _ -> ()

let onReceive a m agents =
  match m with |
  | Signed (m', b) when not(isLogMsg m') ->
  | logRecv a m ;
  | List.iter (fun c ->
    if c <> a then
    send (mkLogMsg !adeqLog) c)
  ) agents |
  | _ -> ()
```

events and the rest of \( \mathbf{es} \) without the log events, or raises an exception if some log event is missing.

We check if an agent is compatible with the DBK pattern using the function `correct`. It takes an agent name \( a \), a set of participating agents \( \text{agents} \) including \( a \) and a trace \( \mathbf{es} \) as inputs. `correct` iterates over the trace using the function `correctIter`, which checks if each event originated by \( a \) is treated as the accounting daemon prescribes by searching for the corresponding log events, transcribing them and checking for adequacy.

The upper bound on the time complexity of checking agent correctness is quadratic in the length of the trace. The function `correct` calls `correctIter` for each event occurring in the trace, and each call to `correctIter` iterates over such events while checking adequacy of logs.

### B.4 A functional implementation of the accounting daemon for DBK

We present a functional implementation of the accounting daemon in Table 10. This definition refines the declarative formulation of the distributed account-
ing daemon shown in Table 6, in particular by implementing the declarative
construct SELECT-SUCH-THAT using a mutable store and its update.

We assume that each agent has a local mutable store adeqLog that is used
to keep the log. An agent can use the functions logSend and logRecv to update
the stored log in order to take into account event sending and receiving, respect-
ively. We assume that agents use the function send m a for sending a message
m to an agent a. In order to avoid logging of logs, as discussed before, we use
messages of a special form for the communication of logs, as facilitated by the
function mkLogMsg.

An agent can implement accountability by executing onSend and onReceive
upon each occurrence of send and receive event respectively.

Figure 1 illustrates the distributed accounting daemon. It presents a trace
in which agents respect the accounting daemon by sending appropriate log mes-
ages. At point 1, the agent a sends a data message to b and logs it by issuing a
logging message at point 2, which is sent to the agent c. Upon reception of the
data message, the agent b issues corresponding log messages, see points 5 and 6.
Note that log messages do not trigger further logging, hence, for example, the
agent c does not send any messages.