Secure Agents

Piero A. Bonatti[*] Sarit Kraus[†] V.S. Subrahmanian[‡]

Abstract

With the rapid proliferation of software agents, there comes an increased need for agents to ensure that they do not provide data and/or services to unauthorized users. We first develop an abstract definition of what it means for an agent to preserve data/action security. Most often, this requires an agent to have knowledge that is impossible to acquire — hence, we then develop approximate security checks that take into account, the fact that an agent usually has incomplete/approximate beliefs about other agents. We develop two types of security checks — static ones that can be checked prior to deploying the agent, and dynamic ones that are executed at run time. We prove that a number of these problems are undecidable, but under certain conditions, they are decidable and (our definition of) security can be guaranteed. Finally, we propose a language within which the developer of an agent can specify her security needs, and present provably correct algorithms for static/dynamic security verification.

[*]Dipartimento di Informatica, Università di Milano, Via Bramante 65, I-26013 Crema, Italy. Email: bonatti@crema.unimi.it

[†]Dept. of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, 52900 Israel, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742 E-Mail: sarit@cs.biu.ac.il

[‡]Institute for Advanced Computer Studies, Institute for Systems Research and Department of Computer Science, University of Maryland, College Park, Maryland 20742. E-mail: vs@cs.umd.edu

Contents

1 Introduction

Over the last few years, there has been intense work in the area of intelligent agents [30, 63]. Applications of such agent technology have ranged from intelligent news and mail filtering programs [40], to agents that monitor the state of the stock market and detect trends in stock prices, to intelligent web search agents [21], to the digital battlefield where agent technology closely monitors and merges information gathered from multiple heterogeneous information sources [1, 35, 36, 52, 61]. More recently, we have seen an increase in the number of agents that automatically interact with one another. Such agents can negotiate with each other, participate in auctions, make group consensus decisions, and the like [34, 60, 46, 32].

In previous work [2, 20, 19], Eiter et. al. have developed a framework for building agents on top of specialized data structures and/or legacy code bases. Each such agent has a "state" and provides a set of services to other agents. Such services include data retrieval services (answering database queries, retrievals from geographic information systems, etc.) as well as computational services (e.g. creating a plan, recognizing features in imagery, finding a route, etc.). However, an agent $\mathtt{a}$ may store a vast quantity of data only some of which it is willing to disclose to another agent $\mathtt{b}$ — for example, a $\mathtt{tank}$ agent may disclose its location only to certain other agents, not all. Likewise, a military route planning agent may create routes only for authorized clients, not for others. In commercial applications, agents may provide data and services only to customers who have paid an appropriate fee. Thus, agent designers need to have a framework within which they can describe what data and what services should be provided by their agent to other agents/clients.

Most existing work on agent security has focused on two aspects — protecting host computers from mobile agents (or applets) [25], and the converse problem of protecting agents from the hosts [47]. Our work complements these two approaches, because (i) we do not restrict interest to mobile agents only but consider the broader class of agents, mobile and otherwise, in multiagent AI environments [42], and (ii) we develop techniques by which an agent can provide services and release data to other agents while maintaining security.

The main contributions and organization of this paper may be summarized as follows.

- In Section 2, we provide a small motivating multiagent example for our security framework.
- In Section 3, we describe a framework called *IMPACT* (Interactive Maryland Platform for Agents Collaborating Together) in which a (very general) concept of agent is introduced [20, 19].
- In Section 4, we provide an abstract concept of an agent that is not dependent upon *IMPACT*, but is applicable to other agent systems as well.
- In Section 5, we introduce two concepts used for defining aspects of agent security — *histories* of what the agent did in the past, and *consequence operations* used by an agent to draw inferences. Intuitively, to prevent agent $\mathtt{b}$ from inferring a secret, agent $\mathtt{a}$ must somehow ensure that agent $\mathtt{b}$'s "true" state of knowledge of the world (which is shaped by agent $\mathtt{b}$'s "true" history and agent $\mathtt{b}$'s "true" consequence operation) does not entail the secret. Similar definitions are needed to ensure that agent $\mathtt{b}$ does not utilize services it is not cleared to use. We also formally define what it means for an agent $\mathtt{a}$ to maintain "true" security" in terms of the above concepts. Specifically,

we show that a naive definition of security called "surface security" is not enough for maintain true security and our notion of data security alleviates this problem. We do likewise for "true" action security.

- It is usually impossible in practice for agent $\mathtt{a}$ to have correct and complete information about agent $\mathtt{b}$'s state, consequence operation and history. Hence, maintaining "true" security is infeasible in practice. To alleviate this problem, we define, in Section 6, what it means for agent $\mathtt{a}$ to approximate $\mathtt{b}$'s state, consequence operation and history. Based on these concepts, we define an approximation of true data security and true action security and show that under certain conditions, approximate security implies true security, i.e. the approximation is "good enough" to maintain true security.
- In Section 7, we show that the general problem of maintaining data and action security for agents is undecidable. It does not matter whether these agents are built in *IMPACT* or in Aglets [54] or in Java[44]. However, this undecidability is also true for approximate data and action security of general agents.
- As a consequence, in Section 8, we provide a (family of) languages through which agent designers can express the security needs of their agents. Using this language, designers of an agent $\mathtt{a}$ can express how agent $\mathtt{a}$ approximates the history, state, consequence operation, etc., of another arbitrary agent $\mathtt{b}$. We show that this language is decidable, and thus provides a polynomially implementable fragment of the general agent security theory proposed in this paper.
- In Section 10, we describe related work on agent security, and assess the strengths and weaknesses of our approach.

2 Motivating Example

Consider a small multiagent application involving two tanks $\mathtt{tank1}$ and $\mathtt{tank2}$, a command center $\mathtt{com\_c}$, and a $\mathtt{monitoring}$ agent.

The two tanks are both engaged in some operational mission, and are continuously aware of their geo-location, bearing, and speed. They are tasked to perform actions by the command center. The command center is authorized to know all information about the tanks.

In contrast, the $\mathtt{monitoring}$ agent may ask the tanks for information on their supply state (e.g. how many rounds of fire/fuel they have, whether any parts need repair, etc.). The $\mathtt{monitoring}$ agent is not authorized to know the precise location of the tank — it is important to note that this does not mean that the tank cannot reveal its bearing/speed to the $\mathtt{monitoring}$ agent. In fact, it may even be able to reveal an old position without compromising security. The $\mathtt{monitoring}$ agent may task the tanks to take appropriate repair actions, but has no authority to change their route, etc.

We will use these agents as a running example throughout this paper.

3 Preliminaries: *IMPACT* Agents

In *IMPACT*, each agent $\mathtt{a}$ is built on top of a body of software code (built in any programming language) that supports a well-defined application programmer interface (either part of the code itself, or developed to augment the code).

Definition 3.1 (Software Code) *We may characterize the code on top of which an agent is built as a triple $\mathcal{S} =_{def} (\mathcal{T}_{\mathcal{S}}, \mathcal{F}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ where:*

1. *$\mathcal{T}_{\mathcal{S}}$ is the set of all data types managed by $\mathcal{S}$,*
2. *$\mathcal{F}_{\mathcal{S}}$ is a set of predefined functions which makes access to the data objects managed by the agent available to external processes, and*
3. *$\mathcal{C}_{\mathcal{S}}$ is a set of type composition operations. A type composition operator is a partial n-ary function c which takes as input types $\tau_1, \ldots, \tau_n$ and yields as a result a type $c(\tau_1, \ldots, \tau_n)$. As c is a partial function, c may only be defined for certain arguments $\tau_1, \ldots, \tau_n$, i.e., c is not necessarily applicable on arbitrary types.*

In general, we will assume that the piece of software $\mathcal{S}^{\mathfrak{a}}$ associated with an agent $\mathfrak{a} \in \mathrm{A}$ is represented by a triple $\mathcal{S}^{\mathfrak{a}} =_{def} (\mathcal{T}^{\mathfrak{a}}_{\mathcal{S}}, \mathcal{F}^{\mathfrak{a}}_{\mathcal{S}}, \mathcal{C}^{\mathfrak{a}}_{\mathcal{S}})$. When we are referring to the code associated with a fixed agent $\mathfrak{a}$, we will often drop the superscript $\mathfrak{a}$ above. Intuitively, $\mathcal{T}_{\mathcal{S}}$ is the set of all data types that are managed by the agent. $\mathcal{F}_{\mathcal{S}}$ intuitively represents the set of all function calls supported by the package $\mathcal{S}$'s application programmer interface (*API*). $\mathcal{C}_{\mathcal{S}}$ the set of ways of creating new data types from existing data types. This characterization of a piece of software code is a well accepted and widely used specification. For example, the *Object Data Management Group*'s *ODMG* standard [13] and the *CORBA* framework [49] are existing industry standards consistent with this specification.

Each agent also has a message box having a well defined set of associated code calls that can be invoked by external programs.

Example 3.1 *Let us assume that the two tank agents each have function calls called:*

- *speed() which returns as output, the current speed (non-negative integer) of the tank;*
- *bearing() which returns as output, the current bearing (integer between 0 and 360) describing the angular bearing of the tank;*
- *location*(`T`) *which returns as output, the pair (x, y) defining the location of the tank at time* `T` *relative to some fixed map;*
- *region*(`T`) *which returns as output, a quadruple (ℓ, r, b, t) describing the region $\{(x', y') | \ell \leq x' \leq r \,\&\, b \leq y' \leq t\}$ such that location*(`T`) $\in$ `region(T)`.

Likewise, the command center agent may support the following function calls:

- *$find\_friendly(\ell, r, b, t)$ which returns as output, the set of all triples containing a friendly tank and its location in the specified region.*
- *$find\_enemy(\ell, r, b, t)$ which returns as output, the set of all triples containing an enemy tank and its location in the specified region.*
- *$distance(x, y, x', y')$ returns as output, the distance between two points.*

The state of an agent, at any given point t in time, consists of the set of all instantiated data objects of types contained in $\mathcal{T}_{\mathcal{S}}^{\mathtt{a}}$:

Definition 3.2 (State of an Agent) *At any given point t in time, the* state of an agent *will refer to a set $\mathcal{O}_{\mathcal{S}}(t)$ of objects from the types $\mathcal{T}_{\mathcal{S}}$, managed by its internal software code. An agent may change its state by taking an action—either triggered internally, or by processing a message received from another agent. Throughout this paper we will assume that except for appending messages to an agent $\mathtt{a}$'s mailbox, another agent $\mathtt{b}$ cannot directly change $\mathtt{a}$'s state. However, it might do so indirectly by shipping the other agent a message issuing a change request.*

Queries and/or conditions may be evaluated against an agent state using the notion of a code call atom and a code call condition defined below.

Definition 3.3 (Code Call/Code Call Atom) *If $\mathcal{S}$ is the name of a software package, f is a function defined in this package, and $(d_1, \dots, d_n)$ is a tuple of arguments of the right input types of f, then $\mathcal{S}:f(\mathtt{d_1}, \dots, \mathtt{d_n})$ is called a* code call.

If `cc` is a code call, and `X` is either a variable symbol, or an object of the output type of `cc`, then $\mathbf{in}(\mathtt{X}, \mathtt{cc})$ *is called a* code call atom.

Code call atoms, when evaluated, return boolean values, and thus may be thought of as special types of logical atoms. Intuitively, a code call atom of the form $\mathbf{in}(\mathtt{X}, \mathtt{cc})$ succeeds just in case when `X` can be set to a pointer to one of the objects in the set of objects returned by executing the code call. In database terminology, `X` is a cursor on the result of executing the code call.

Definition 3.4 (Code Call Condition) *A* code call condition χ *is defined as follows:*

1. *Every code call atom is a code call condition.*
2. *If $\mathtt{s}, \mathtt{t}$ are either variables or objects, then $\mathtt{s} = \mathtt{t}$ is a code call condition.*
3. *If $\mathtt{s}, \mathtt{t}$ are either integers/real valued objects, or are variables over the integers/reals, then $\mathtt{s} < \mathtt{t}, \mathtt{s} > \mathtt{t}, \mathtt{s} \geq \mathtt{t}, \mathtt{s} \leq \mathtt{t}$ are code call conditions.*
4. *If χ_1, χ_2 are code call conditions, then $\chi_1 \,\&\, \chi_2$ is a code call condition.*

A code call condition satisfying any of the first three criteria above is an atomic *code call condition.*

Example 3.2 *Let us return to the case of example 3.1. Here are some example code call conditions.*

1. $\mathbf{in}(\mathtt{X}, \mathtt{tank1} : \mathit{speed}()) \,\&\, X \geq 20$.
   This code call condition succeeds iff the speed of $\mathtt{tank1}$ *exceeds 20 units.*
2. $\mathbf{in}(\mathtt{X}, \mathtt{tank1} : \mathit{speed}()) \,\&\, \mathbf{in}(\mathtt{Y}, \mathtt{tank2} : \mathit{speed}()) \,\&\, X > Y$.
   This code call condition succeeds iff the speed of $\mathtt{tank1}$ *exceeds that of* $\mathtt{tank2}$.

3. $\mathbf{in}(\mathtt{V}, \mathtt{com\_c}: \mathit{find\_friendly}(\mathtt{10,20,10,20}))\,\&\,\mathbf{in}(\mathtt{V'}, \mathtt{com\_c}: \mathit{find\_enemy}(\mathtt{10,20,10,20}))\,\&\,\mathbf{in}(\mathtt{D}, \mathtt{com\_c}: \mathit{distance}(\mathtt{V.x}, \mathtt{V.y}, \mathtt{V'.x}, \mathtt{V'.y}))\,\&\,D < 5$.
This code call condition finds all pairs (V, V') of tanks where V is friendly, V' is an enemy tank, and the two tanks are within 5 units of distance from each other. Such a code call condition may be used by someone monitoring battlefield conditions who wants to find all tanks V which are in danger.

It is important to note that not all code call conditions are evaluable. Subrahmanian et. al. [53, ch. 4] identify a class of code call conditions called *safe* code call conditions and show that for each code call condition which is safe, there is at least one evaluation order that can be used to evaluate the individual conjuncts in the code call condition.

Each agent has an action-base consisting of a description of the various actions that the agent is capable of executing. Actions change the state of the agent and perhaps the state of other agents' `msgboxes`. Such actions comprise the services that other agents might request.

An agent also has an associated *notion of concurrency* which takes a set of actions and the agent state as input, and merges the actions into a single "unified" action that is executed in lieu of the set of individual actions. [20] provide several alternative implementations of such notions of concurrency – the agent developer selects or defines one that is appropriate for his agent.

Each agent has an associated set of integrity constraints—only states that satisfy these constraints are considered to be *valid* or *legal* states. Each agent has an associated set of action constraints that define the circumstances under which certain actions may be concurrently executed. As at any given point t in time, many sets of actions may be concurrently executable, each agent has an *Agent Program* that determines what actions the agent can take, what actions the agent cannot take, and what actions the agent must take. The agent program is defined as follows.

Definition 3.5 (Status Atom/Status Set) *If $\alpha(\vec{t})$ is an action, and $Op \in \{\mathbf{P}, \mathbf{F}, \mathbf{W}, \mathbf{Do}, \mathbf{O}\}$, then $Op\alpha(\vec{t})$ is called a* status atom. *A* status set *is a finite set of status atoms.*

Intuitively, $\mathbf{P}\alpha$ means α is permitted, $\mathbf{F}\alpha$ means α is forbidden, $\mathbf{O}\alpha$ means α is obligatory, $\mathbf{Do}\,\alpha$ means α is actually done, and $\mathbf{W}\alpha$ means that the obligation to perform α is waived.

Definition 3.6 (Agent Program) *An agent program $\mathcal{P}$ is a finite set of rules of the form*

$$A \;\leftarrow\; \chi \,\&\, \pm A_1 \,\&\, \ldots \,\&\, \pm A_n$$

where χ is a code call condition and $A_1, \ldots, A_n$ are status atoms.

As is common in logic programming, all variables are assumed to be implicitly universally quantified at the front of the rule. As a consequence, it is important to note that the same variable may occur both inside χ and inside the status atoms $A_1, \ldots, A_n$ — such variables express conditions spanning both the code call condition and the status atoms.

The semantics of agent programs are well described in [20, 19]. Appendix A provides a brief overview of the three most popular semantics for agent programs.

In order to assist the reader, Table 1 lists the notation used in this paper, and the section in which each is first defined.

Notation	Location	Description
$\mathcal{S}$	Def. 3.1	Software code
$\mathcal{O}_{\mathcal{S}}(t)$	Def. 3.2	Agent state
$\mathcal{S}:f(\mathtt{d}_1,\dots,\mathtt{d}_n)$	Def. 3.3	Code call
$\mathbf{in}(\mathtt{X},\mathcal{S}:f(\mathtt{d}_1,\dots,\mathtt{d}_n))$	Def. 3.3	Code call atom
χ	Def. 3.4	Code call condition
$\alpha(\vec{t})$	Section 3	Action
$\mathcal{L}_{\mathfrak{a}}$	Section 4.2 beginning	fact language of agent $\mathfrak{a}$
h	Def. 4.1	History
$\text{pos}\mathcal{H}_{\mathfrak{a}}$	Def. 4.2	Possible histories of $\mathfrak{a}$
$\text{Cn}_{\mathfrak{a}}$	Def. 4.4	consequence operation of $\mathfrak{a}$
$\vdash_{\mathfrak{a}}$	Section 4.3 after Def. 4.4	provability relation
$Sec_{\mathfrak{a}}$	Def. 5.1	Agent secrets function
$ASec_{\mathfrak{a}}$	Def. 5.2	Agent action security function
$h_1 \overset{\mathfrak{ab}}{\longleftrightarrow} h_2$	Def. 5.4	Compatible histories
$\mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}})$	4.3	$\mathfrak{b}$'s state at $h_{\mathfrak{b}}$
$\mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}})$	Def. 5.5	Violated secrets
$\text{pos}\mathcal{H}_{\mathfrak{b}}^{\mathfrak{a}}$	Def. 6.1	Possible histories approximation
$\leadsto_{\mathsf{h}}$	Def. 6.2	History correspondence relation
$AppH_{\mathfrak{b}}(h)$	Def. 6.3	Approximate current history
$App\mathcal{L}_{\mathfrak{b}}$	Def. 6.4	Approximate fact language
$\leadsto_{\mathsf{f}}$	Def. 6.5	Fact correspondence relation
$\leadsto_{\mathsf{c}}$	Def. 6.7	Condition correspondence relation
$App\mathcal{O}_{\mathfrak{b}}$	Def. 6.8	Approximate state function
$AppSec(\mathfrak{b})$	Def. 6.10	Approximate secrets
$AppCn_{\mathfrak{b}}$	Def. 6.12	Approximate consequence operation
$\mathsf{OViol}_{\mathfrak{b}}$	Def. 6.17	Overestimate of violated secrets
$\mathsf{UViol}_{\mathfrak{b}}$	Def. 6.18	Underestimate of violated secrets
$G_0 \longmapsto_R^{\theta} G_m$	Def. 8.7	Pseudo-derivation

Table 1: Summary of notation

4 Abstract Agents

As described in the Introduction, each agent has a "true" history (describing its past interactions with other agents), and a "true" consequence operation. In addition, a logical notion of state built on top of the previous definition will be needed to define what an agent knows at a given instant of time. Intuitively, to preserve security, we need to ensure that no secret is known to the agent.

4.1 Abstract Behavior: Histories

There are two types of events that may determine an agent a's behavior. An *action event* $\langle \alpha(\vec{t}), \mathsf{b} \rangle$ describes an action that a has taken in response to a request by an agent b. If $\mathsf{b} = \mathsf{a}$, then $\alpha(\vec{t})$ is a "spontaneous" action, executed to achieve some of a's own goals. A *message event* is represented as a triple of the form $\langle \mathit{sender}, \mathit{receiver}, \mathit{body} \rangle$, where *sender* and *receiver* are agents, $\mathit{sender} \neq \mathit{receiver}$, and *body* is either a service request ϱ or an *answer*, that is, a set of ground code call atoms.

Definition 4.1 (Histories) *A* history *is a possibly infinite sequence of events, such as* $\langle e_1, e_2, \dots \rangle$. *We say that a history h is a* history for a *if each action in h can be executed by a, and for all messages $\langle s, r, m \rangle$ in h, either $s = \mathsf{a}$ or $r = \mathsf{a}$.*

The concatenation *of two histories h_1 and h_2 will be denoted by $h_1 \cdot h_2$. With a slight abuse of notation, we shall sometimes write $h \cdot e$, where e is an event, as an abbreviation for the concatenation $h \cdot \langle e \rangle$.*

A history for a keeps track of a set of messages that a has exchanged with other agents, and a set of actions that a has performed.

The notion of history for a captures histories that are *syntactically* correct. However, not every history for a describes a possible behavior of a. For instance, some histories are impossible because a's code will never lead to that sequence of events. Some others are impossible because they contain messages coming from agents that will never want to talk to a. This leads to the notion of "possible histories" below.

Definition 4.2 (Possible Histories) *Every agent a has an associated set of histories, $pos\mathcal{H}_{\mathsf{a}}$, called the* possible histories *of agent a.*

For example, a history where agent a sends mail to agent b without a prior request may not constitute a possible history for agent b.

Example 4.1 *A possible history for* $\mathsf{tank1}$ *agent may have the form* $\langle \dots e_1, e_2, e_3, e_4 \dots \rangle$, *where:*

$$\begin{aligned}
e_1 &= \langle \mathtt{com\_c}, \mathsf{tank1}, \mathit{set{:}speed}(\mathit{new\_speed}) \rangle, \\
e_2 &= \langle \mathit{set\_speed}(55kmh), \mathtt{com\_c} \rangle, \\
e_3 &= \langle \mathtt{com\_c}, \mathsf{tank1}, \mathit{location}(\mathtt{X}_{now}) \rangle, \\
e_4 &= \langle \mathsf{tank1}, \mathtt{com\_c}, \{\mathbf{in}((50,20,40), \mathsf{tank1} : \mathit{location}(\mathtt{X}_{now}))\} \rangle .
\end{aligned}$$

Notice that e_1, e_3, e_4 *have three arguments and hence they are message events.* e_1 *refers to a message sent by* $\mathsf{com\_c}$ *to* $\mathsf{tank1}$ *requesting that the* $\mathsf{tank1}$*'s speed be set to a new speed.* e_2 *has only two arguments and hence describes an action — in this case, the event says that the speed has been set to 55 in response to* $\mathsf{com\_c}$*'s request. Event* e_3 *is another message event sent by* $\mathsf{com\_c}$ *to* $\mathsf{tank1}$ *requesting location information. Event* e_4 *is another message event sent by* $\mathsf{tank1}$ *to* $\mathsf{com\_c}$ *with the desired information.*

4.2 Logical Agent States

The state of an agent may be *represented* as the set of all ground code call atoms $\mathbf{in}(\mathtt{o}, \mathcal{S}: f(\mathtt{a}_1, \ldots, \mathtt{a}_n))$ which are true in the state, where $\mathcal{S}$ is the name of a data structure manipulated by the agent, and f is one of the functions defined on this data structure. Each of these ground code call atoms may be thought of as a logical atom. For any given agent a, the set of ground code call atoms that can be used by a will be denoted by $\mathcal{L}_{\mathsf{a}}$, and will be called the *fact language of* a .

Example 4.2 *Returning to example 3.2, the state of the* $\mathsf{tank1}$ *agent may consist of the ground code call atoms:*

$$\mathbf{in}((5,5), \mathsf{tank1}: \mathit{location}(\mathtt{X}_{\mathit{now}})).$$

$$\mathbf{in}(25, \mathsf{tank1}: \mathit{speed}()).$$

$$\mathbf{in}(120, \mathsf{tank1}: \mathit{bearing}()).$$

Clearly, the state of a at a given point in time is determined by the history of a up to that point. Therefore, it is natural to model a's state changes as a function from histories to states. This is done in the next definition.

Definition 4.3 (Agent State at h: $\mathcal{O}_{\mathsf{a}}(h)$) *For all agents* a *and all histories* h *for* a, *we denote by* $\mathcal{O}_{\mathsf{a}}(h)$, *the state of* a *immediately after the sequence of events* h. *The initial state of* a *(i.e. the state of* a *when it was initially deployed) is denoted by* $\mathcal{O}_{\mathsf{a}}(\langle\rangle)$.

4.3 Agent Consequence Operation

In principle, "intelligent" agents can derive new facts from the information explicitly stored in their state. Different agents have different reasoning capabilities. Some agents may perform no reasoning on the data they store, some may derive new information using numeric calculations, while others may have sophisticated inference procedures.

Definition 4.4 (Agent Consequence Operation) *We assume that each agent* a *has an associated consequence operation* Cn_{a}, *that takes as input, a set of ground code call atoms, and returns as output, a set of ground code call atoms.* $Cn_{\mathsf{a}}(F)$ *returns as output, all ground code call atoms implied by the input set* F, *according to the notion of consequence adopted by* a. Cn_{a} *is required to satisfy the well acceptable general axioms:*

1. **Inclusion** $Cn_{\mathfrak{a}}(X) \supseteq X$ *;*

2. **Idempotence** $Cn_{\mathfrak{a}}(Cn_{\mathfrak{a}}(X)) = Cn_{\mathfrak{a}}(X)$ *.*

Our definition of agent consequence builds upon the classical notion of an abstract consequence operation, originally proposed by [56]. Almost all standard provability relations, $\vdash$, for different proof systems ranging from classical logic to modal logics to multivalued logics, induce a function $\mathrm{Cn}^{\vdash}$ as follows:

$$\mathrm{Cn}^{\vdash}(X) \quad =_{def} \quad \{\psi \mid X \vdash \psi\}\,.$$

Conversely, each abstract consequence operation $\mathrm{Cn}_{\mathfrak{a}}$ induces a provability relation

$$S \vdash_{\mathfrak{a}} \psi \quad \text{if, by definition,} \quad \forall X : S \subseteq X \subseteq \mathcal{L}_{\mathfrak{a}},\ \psi \in \mathrm{Cn}_{\mathfrak{a}}(X)\,.$$

Note a subtle difference between $\vdash_{\mathfrak{a}}$ and $\mathrm{Cn}_{\mathfrak{a}}$: in $S \vdash_{\mathfrak{a}} \phi$, S is treated as a *partial* description of a state X, while the argument X of $\mathrm{Cn}_{\mathfrak{a}}$ is taken as a *complete* description of $\mathfrak{a}$'s state.

It is also important to note that agent consequence operations are not required to be *sound* with respect to classical logic. This is because some agents may make decisions on the basis of conditions that *normally* or *plausibly* hold; the consequence operation of such agents is in general not a subset of classical inferences. Moreover, drawing conclusions requires resources; some agents may want to infer all valid conclusions from their state, while others may only draw inferences obtainable through a bounded number of inferences. This explains why agent consequence operations are not required to be *complete* w.r.t. classical inference (i.e. agent consequence operations may not include all classical inferences).

Example 4.3 *Returning to example 4.2 where* tank1*'s state can be viewed as a set of first-order formulas (the code call conditions which are true in the state). Then,* tank1 *may be able to infer from these first-order formulas (some) logical consequences, using the standard inferences of first-order logic.*

5 Security of Abstract Agents

In this section, we show how we may build a notion of security on top of the abstract definition of agents given earlier.

- First, in Section 5.1 we will describe, for each agent $\mathfrak{a}$, what data and actions it wishes to protect from another agent $\mathfrak{b}$. When handling a service request, agent $\mathfrak{a}$ must ensure that such data is not disclosed to agent $\mathfrak{b}$, and such actions are not executed on behalf of agent $\mathfrak{b}$.
- In Section 5.2, we will define what it means for an agent to preserve security, with respect to the security specifications introduced in Section 5.1.
- Finally, in Section 5.3, *maximally cooperative histories* will be introduced. The underlying idea is that in many cases, we want security-preserving agent services to be as close as possible to the unrestricted (non-security-preserving) services, i.e. $\mathfrak{a}$'s behavior should be distorted as little as possible when attempting to maintain security.

5.1 Security Specifications

In this section, we define what kinds of *data* an agent would like to protect from another agent, and also what kinds of *actions* an agent would like to avoid executing for other agents.

Definition 5.1 (Agent Secrets Function $Sec_{\mathtt{a}}$) *Suppose* $\mathtt{a}$ *is an agent.* $Sec_{\mathtt{a}}$ *is a function which associates with any other agent* $\mathtt{b} \neq \mathtt{a}$, *a set of ground code call atoms which* $\mathtt{a}$ *would like to keep secret from* $\mathtt{b}$.

Intuitively, $\mathtt{a}$ would like to prevent $\mathtt{b}$ from *inferring* the ground code call atoms in $Sec_{\mathtt{a}}(\mathtt{b})$.

Example 5.1 *In the scenario of the tanks we assumed that the* monitor *agent is not allowed to know the tanks' locations. Thus,* tank1 *agent should have an associated secrets function* $Sec_{\mathtt{tank1}}$ *such that all the facts* $\mathbf{in}(x, \mathtt{tank1} : location(\mathtt{X}_{now}))$ *should be contained in* $Sec_{\mathtt{tank1}}(\mathtt{monitor})$.

The concept of an agent action security function describes what actions an agent may or may not perform for another agent.

Definition 5.2 (Agent Action Security Function $ASec_{\mathtt{a}}$) *An agent action security function associated with agent* $\mathtt{a}$ *is a function* $ASec_{\mathtt{a}}$ *that associates with any other agent* $\mathtt{b} \neq \mathtt{a}$, *a set consisting of (i) outgoing request messages of the form* $\langle \mathtt{a}, \mathtt{c}, \varrho \rangle$ $(\mathtt{c} \neq \mathtt{b})$, *and (ii) sequences of ground action names.*

Roughly speaking, $ASec_{\mathtt{a}}(\mathtt{b})$ contains a set of forbidden action sequences that $\mathtt{a}$ does not want to execute upon $\mathtt{b}$'s requests. It also includes requests that $\mathtt{a}$ is not willing to issue on behalf of $\mathtt{b}$.

Example 5.2 *As mentioned in Section 2, the* monitor*ing agent may task the tanks to take appropriate repair actions, but has no authority to change their route, etc.*

Thus, $ASec_{\mathtt{tank1}}(\mathtt{monitor})$ *should contain (among other sequences) all the simple sequences*
$\langle set\_speed(x) \rangle$ ... $\langle move\_to(y) \rangle$ ... *etc.*

In some cases, the set $ASec_{\mathtt{a}}(\mathtt{b})$ may be closed under action equivalence. For example, suppose there exist two actions `printf(s)` and `fprintf(stdout,s)` that execute the C functions associated with these names. These two actions are equivalent, and hence if $\alpha_1, \alpha_2, \ldots, \alpha_9$ is a forbidden action sequence and $\alpha_2 = \mathtt{printf}(\mathtt{s})$, then the action sequence $\alpha_1, \mathtt{fprintf}(\mathtt{stdout}, \mathtt{s}), \alpha_3, \ldots, \alpha_9$ should also be forbidden.

One may therefore wonder whether we should insist that if an action sequence is in $ASec_{\mathtt{a}}(\mathtt{b})$, then every action sequence equivalent to it should also be in $ASec_{\mathtt{a}}(\mathtt{b})$. Using the real world operation of computer systems as a guide, the answer seems to be "no." To see why, consider simple email. A user may write on another user's mailbox file only through certified e-mail programs. No sequence of individual `open`, `close`, `read` and `write` operations is admitted on another user's mailbox, although some of these sequences update the mailbox exactly as the e-mail program would. Accordingly, $ASec_{\mathtt{a}}(\mathtt{b})$ need not necessarily be closed under action equivalence.

5.2 Secure Histories

What does it mean for an agent to preserve security? A full answer to this question must deal both with the protection of agents' *data*, and with restrictions on the *actions* that agents may execute in response to incoming requests.

Let us consider data protection first. Standard approaches require systems (be they agents, databases or other packages) to include no secrets in their answers. This is definitely a reasonable security requirement, that we call *surface security.*

Recall that pos$\mathcal{H}_{\mathsf{a}}$ denotes the set of *all possible histories* for an agent a (i.e. the possible behaviors of a).

Definition 5.3 (Surface Security) *A history $h_{\mathsf{a}} \in pos\mathcal{H}_{\mathsf{a}}$ is* surface secure w.r.t. b *if for all messages $\langle \mathsf{a}, \mathsf{b}, Ans \rangle$ in h_{a},*

$$Ans \cap Sec_{\mathsf{a}}(\mathsf{b}) = \emptyset \,.$$

If all histories $h_{\mathsf{a}} \in pos\mathcal{H}_{\mathsf{a}}$ are surface secure w.r.t. b then we say that agent a is surface secure w.r.t. b.

Example 5.3 *In the scenario of the tanks, we assumed that the* monitor *agent is not allowed to know the tanks' locations. Thus, a history in which* tank1 *does not explicitly tell the* monitor *agent its location will be surface secure. However, the* monitor *agent may still deduce the location. For example, if it knows that* tank1 *has been moving at a constant speed d, along a given bearing b for the last 30 minutes, it can derive the current position of the tank from its location at time $t = \texttt{now} - 30$. Note, that in this example, the tank's position 30 minutes ago—although not a secret in itself—suffices to let the* monitor *agent infer a secret (the current position of the tank).*

In another example, the monitor *agent may be able to deduce* tank1*'s location from knowing that it is low in fuel.*[1] *In this example, the* tank1*'s being low in fuel may lead to violating of a secret even though it is not a secret in itself.*

Although this somewhat minimal form of security may be satisfactory against simple client agents, it doesn't guarantee data protection from smart agents because such agents can derive new information through their consequence operation; surface security does not verify that no secret be *derived* through the consequence operation.

A naive approach to this problem consists of stating that *an agent a is data secure if its client agents can never deduce any secret.* However, this definition does not take into account the fact that security breaches might be caused by some other agent $\mathsf{c} \neq \mathsf{a}$. The problem is that b might come to know some secret s because it was told this by c. Clearly, agent a has in no way caused security to be violated in this situation. Under the naive definition, a would not be data-secure simply because c disclosed s. This would happen even in the extreme case where a never answers incoming requests and maintains perfect silence!

This paradoxical situation can be avoided by adopting a more realistic notion of security. The underlying intuition is that agents are responsible only for their own answers. Roughly

[1]In real scenarios the monitor agent will need more information, e.g., the region where tank1 is located, to conclude tank1's location. However, we make this assumption to simplify the discussion.

speaking, an agent can be said to be secure if its answers never *increase* the set of secrets known by other agents. With respect to the previous example, a should be regarded as data secure as long as b cannot derive new secrets using a's answers. To state this formally, we need a couple of intermediate definitions.

Definition 5.4 (Compatible Histories $h_1 \xleftrightarrow{\mathsf{ab}} h_2$) *Let a and b be agents. We say that two histories h_1 and h_2 are* ab-compatible, *denoted $h_1 \xleftrightarrow{\mathsf{ab}} h_2$, if the subsequences of h_1 and h_2 obtained by removing all events but the messages of the form $\langle \mathsf{a}, \mathsf{b}, \dots \rangle$ and $\langle \mathsf{b}, \mathsf{a}, \dots \rangle$ are equal. Furthermore, if $h_1 \xleftrightarrow{\mathsf{ab}} h_2$ and the last events of h_1 and h_2 are the same, then we write $h_1 \overset{\mathsf{ab}}{\Longleftrightarrow} h_2$, and say that h_1, h_2 are* strongly ab*-compatible.*

Intuitively, histories h_1 and h_2 are ab-compatible iff the two histories are identical as far as messages between the agents a, b are concerned. Therefore, h_1 and h_2 might be a's and b's view (respectively) of the same global sequence of events. Note that h_1 and h_2 may differ on interactions involving agents other than a, b, but they are considered to be a, b compatible if they coincide on events involving a, b.

Example 5.4 *Consider the two histories h_1, h_2 given below.*

$$\begin{aligned} h_1 &= \langle \mathsf{b}, \mathsf{a}, \varrho_1 \rangle, \langle \mathsf{a}, \mathsf{c}, \varrho_2 \rangle, \langle \mathsf{c}, \mathsf{b}, \varrho_3 \rangle, \langle \mathsf{a}, \mathsf{b}, \mathit{ans}_1 \rangle. \\ h_2 &= \langle \mathsf{b}, \mathsf{a}, \varrho_1 \rangle, \langle \mathsf{a}, \mathsf{c}, \varrho_4 \rangle, \langle \mathsf{c}, \mathsf{b}, \varrho_3 \rangle, \langle \mathsf{a}, \mathsf{b}, \mathit{ans}_1 \rangle. \end{aligned}$$

It is easy to see that histories h_1, h_2 are ab-compatible and bc-compatible, but they are not ac-compatible. Furthermore, h_1 and h_2 are strongly ab-compatible and strongly bc-compatible, as the last events of these two histories are identical.

In addition to the notion of compatible histories, we need a concise notation for the set of secrets of a that can be violated (i.e. inferred) by b at some point in time, corresponding to history h_b. Recall that we use $\mathcal{O}_\mathsf{b}(h_\mathsf{b})$ to denote b's state at h_b, and that $\mathrm{Cn}_\mathsf{b}(\mathcal{O}_\mathsf{b}(h_\mathsf{b}))$ is the set of facts that can be derived by b from that state.

Definition 5.5 (Violated Secrets) $\mathsf{Violated}^{\mathsf{a}}_{\mathsf{b}}(h_\mathsf{b}) = \mathit{Cn}_\mathsf{b}(\mathcal{O}_\mathsf{b}(h_\mathsf{b})) \cap \mathit{Sec}_\mathsf{a}(\mathsf{b})$.

Example 5.5 *In the scenario of the tanks we assumed that the* $\mathsf{monitor}$ *agent is not allowed to know the tanks' locations. A possible history h_{monitor} for the* $\mathsf{monitor}$ *agent may have the form $\langle \dots e_1, e_2, \dots \rangle$, where:*

$$\begin{aligned} e_1 &= \langle \mathsf{monitor}, \mathsf{tank1}, \mathit{fuel\_level}() \rangle, \\ e_2 &= \langle \mathsf{tank1}, \mathsf{monitor}, \{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\} \rangle. \end{aligned}$$

Suppose that after h_{monitor}, the $\mathsf{monitor}$ *agent's state (with respect to* $\mathsf{tank1}$*) only includes the fuel level of* $\mathsf{tank1}$*, and suppose that the* $\mathsf{monitor}$ *agent cannot deduce anything new from this fact. Then,* $\mathsf{Violated}^{\mathsf{tank1}}_{\mathsf{monitor}}(h_{\mathsf{monitor}})$ *is empty.*

However, if from knowing that the fuel level of $\mathsf{tank1}$ *is low, the* $\mathsf{monitor}$ *agent can conclude that* $\mathsf{tank1}$ *is in the support center, e.g., given that the location of the support* *center is* (50, 20, 40) *it may conclude that* $\mathbf{in}((50,20,40), \mathsf{tank1} : \mathit{location}(\mathtt{X}_{now}))$ *and if this is the actual location of* tank1*, then*

$$\mathsf{Violated}^{\mathsf{tank1}}_{\mathsf{monitor}}(h_{\mathsf{monitor}}) = \{\mathbf{in}((50,20,40), \mathsf{tank1} : \mathit{location}(\mathtt{X}_{now}))\}$$

We are now ready to formalize the important concept of data security, which says that for an agent to be data secure, it must guarantee that it will never increase the set of secrets violated by another agent.

Definition 5.6 (Data Security) *A history* $h_{\mathsf{a}} \in pos\mathcal{H}_{\mathsf{a}}$ *is* data secure w.r.t. b *if for all prefixes* $h \cdot e$ *of* h_{a} *such that* e *is an answer message* $\langle \mathsf{a}, \mathsf{b}, Ans\rangle$*, and for all histories* $h_{\mathsf{b}} \cdot e \in pos\mathcal{H}_{\mathsf{b}}$ *such that* $h_{\mathsf{b}} \cdot e \overset{\mathsf{ab}}{\Longleftrightarrow} h \cdot e$*,*

$$\mathsf{Violated}^{\mathsf{a}}_{\mathsf{b}}(h_{\mathsf{b}}) \supseteq \mathsf{Violated}^{\mathsf{a}}_{\mathsf{b}}(h_{\mathsf{b}} \cdot e)\,.$$

If all histories $h_{\mathsf{a}} \in pos\mathcal{H}_{\mathsf{a}}$ *are data secure w.r.t.* b *then we say that* a *is data secure w.r.t.* b*.*

To understand this definition, recall that the conditions $h_{\mathsf{b}} \cdot e \in \mathrm{pos}\mathcal{H}_{\mathsf{b}}$ and $h_{\mathsf{b}} \cdot e \overset{\mathsf{ab}}{\Longleftrightarrow} h \cdot e$, state that $h_{\mathsf{b}} \cdot e$ is a possible history for b when a's answer reaches b. The inclusion in Definition 5.6 says that the set of violated secrets (of b) does not increase after receiving a's answer. By quantifying over all possible histories $h_{\mathsf{b}} \cdot e$ with the aforementioned properties, we require data to be protected no matter what actions b may take before getting the answer, possibly including sending requests to other agents and getting their answers.

Example 5.6 *We return to the the scenario of Example 5.5 and consider the history* $h_{\mathsf{tank1}} = \langle \ldots e_1, e_2, \ldots \rangle$ *where* e_1 *and* e_2 *are as in Example 5.5. Suppose further that all the histories in* $pos\mathcal{H}_{\mathsf{monitor}}$ *that includes* e_1 *and* e_2 *includes the additional event*

$$e_0 = \langle \mathsf{com\_c}, \mathsf{monitor}, \{\mathbf{in}((50,20,40), \mathsf{tank1} : \mathit{location}(\mathtt{X}_{now}))\}\rangle$$

which occurred before e_1*. Even though, as in the previous example,* $\mathsf{Violated}^{\mathsf{tank1}}_{\mathsf{monitor}}(h_{\mathsf{monitor}}) = \{\mathbf{in}((50,20,40), \mathsf{tank1} : \mathit{location}(\mathtt{X}_{now}))\}$*,* h_{tank1} *is data secure. Intuitively, this happens because the location of* tank1 *has been originally revealed to the* monitor *agent by the* com_c *agent and so* tank1*'s answer does not lead to the revelation of any new secrets.*

Interestingly enough, the definition of data security encompasses the case (corresponding to the strict inclusion $\mathsf{Violated}^{\mathsf{a}}_{\mathsf{b}}(h_{\mathsf{b}}) \supset \mathsf{Violated}^{\mathsf{a}}_{\mathsf{b}}(h_{\mathsf{b}} \cdot e)$) in which a convinces b that some previously violated secret does not hold—although in practice this may be just as hard to do as it is desirable.

In general, the notions of surface security and data security are incomparable, in the sense that neither of them implies the other. For example, as we have already pointed out, surface security does not prevent client agents from inferring secrets, so surface security does not imply data security. Conversely, data security does not always entail surface security. For example, if a sends b secrets only when b already knows them (a game well-known by double-crossers), then data security is enforced, while surface security is violated. However, as stated in the following theorem, in some cases, surface security entails data security. Roughly speaking, when agents make no deductions (i.e., their consequence operation is the identity function), surface security suffices to guarantee data security.

Theorem 5.1 *Suppose the consequence operation of* $\mathfrak{b}$*,* $Cn_{\mathfrak{b}}$*, is the identity function, and suppose that for all histories* $h_{\mathfrak{b}} \cdot e$ *such that* $e = \langle \mathfrak{a}, \mathfrak{b}, Ans \rangle$*, the new state of* $\mathfrak{b}$ *is*

$$\mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}} \cdot e) =_{def} \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}}) \cup Ans \,.$$

Then, each surface secure history $h_{\mathfrak{a}}$ *for* $\mathfrak{a}$ *is data secure w.r.t.* $\mathfrak{b}$*.*

Proof: Suppose not. Then for some prefix $h' \cdot e$ of some surface secure history $h_{\mathfrak{a}}$ s.t. (such that) $e = \langle \mathfrak{a}, \mathfrak{b}, Ans \rangle$, for some history $h_{\mathfrak{b}} \cdot e \overset{\mathfrak{ab}}{\Longleftrightarrow} h' \cdot e$ and for some secret $f \in Sec_{\mathfrak{a}}(\mathfrak{b})$, we have

$$f \in \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}} \cdot e) \setminus \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}}) \,.$$

Moreover, by definition of $\mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}$ and by the hypothesis on $\mathrm{Cn}_{\mathfrak{b}}$,

$$\mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}} \cdot e) \setminus \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}}) = (\mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}} \cdot e) \cap Sec_{\mathfrak{a}}(\mathfrak{b})) \setminus (\mathcal{O}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}}) \cap Sec_{\mathfrak{a}}(\mathfrak{b})) \,.$$

From the other hypothesis, it follows that

$$(\mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}} \cdot e) \cap Sec_{\mathfrak{a}}(\mathfrak{b})) \setminus (\mathcal{O}_{\mathfrak{b}}^{\mathfrak{a}}(h_{\mathfrak{b}}) \cap Sec_{\mathfrak{a}}(\mathfrak{b})) \subseteq Ans \cap Sec_{\mathfrak{a}}(\mathfrak{b}) \,.$$

We conclude that $f \in Ans \cap Sec_{\mathfrak{a}}(\mathfrak{b})$. This implies that $h_{\mathfrak{a}}$ is *not* surface secure; a contradiction. ∎

Moreover, if we further assume that the client agent $\mathfrak{b}$ does not store any answer coming from agents other than $\mathfrak{a}$, then surface security and data security coincide. We use a particular instance of such agents to prove the following statement, that will be needed in several proofs in the rest of the paper.

Proposition 5.2 (Data Security vs. Surface Security) *There exist multi-agent systems where surface security coincides with data security.*

Proof: Consider a simple multi-agent system consisting of two agents $\mathfrak{a}$ and $\mathfrak{b}$. Let $\mathfrak{a}$'s and $\mathfrak{b}$'s possible histories have the form

$$h_n = \langle q_1, a_1, \dots, q_n, a_n \rangle,$$

where each q_i is a request message from $\mathfrak{b}$ to $\mathfrak{a}$, and each a_i is $\mathfrak{a}$'s answer to q_i, i.e. a message of the form $\langle \mathfrak{a}, \mathfrak{b}, Ans_i \rangle$. Let $\mathcal{O}_{\mathfrak{b}}(\langle\rangle) =_{def} \emptyset$, and

$$\mathcal{O}_{\mathfrak{b}}(h_n) =_{def} \bigcup_{i=1}^{n} Ans_i \,.$$

Finally, let $\mathrm{Cn}_{\mathfrak{b}}$ be the identity function over $\mathfrak{b}$'s states. Then $\mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(\langle\rangle) = \emptyset$, and

$$\begin{aligned} & h_n \text{ is data secure} \\ & \quad \textit{iff} \;\; \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(\langle\rangle) \supseteq \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_1) \supseteq \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_2) \supseteq \dots \supseteq \mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_n) \end{aligned}$$

Moreover, $\mathsf{Violated}_{\mathfrak{b}}^{\mathfrak{a}}(h_n) = \bigcup_{i=1}^{n} Ans_i \cap Sec_{\mathfrak{a}}(\mathfrak{b})$, by definition of $\mathcal{O}_{\mathfrak{b}}$ and $\mathrm{Cn}_{\mathfrak{b}}$; therefore

$$\mathfrak{a} \text{ is data secure} \quad \textit{iff} \;\; \forall i : 1 \le i \le n,\; Ans_i \cap Sec_{\mathfrak{a}}(\mathfrak{b}) = \emptyset \,.$$

But this is equivalent to saying that $\mathfrak{a}$ is data secure iff $\mathfrak{a}$ is surface secure. ∎

The notion of data security above may be extended to the case of action security as shown below.

Definition 5.7 (Action Security) *Let $h_{\mathsf{a}} \in pos\mathcal{H}_{\mathsf{a}}$ and let $\mathsf{act}(h_a, \mathsf{b})$ be the subsequence of h_{a} consisting of all the actions $\langle \alpha, \mathsf{b} \rangle$ done for* b*. We say that h_{a} is* action secure w.r.t. b *if $\mathsf{act}(h_a, \mathsf{b})$ contains no sequence from $ASec_{\mathsf{a}}(\mathsf{b})$.*

If all histories $h_{\mathsf{a}} \in pos\mathcal{H}_{\mathsf{a}}$ are action secure w.r.t. b*, then we say that* a *is action secure w.r.t.* b*.*

Example 5.7 *Suppose the* monitor *agent has no authority to change* tank1*'s speed. In this case, the following possible history h_{tank1} for* tank1 *is* not *action secure w.r.t.* monitor*: $h_{\mathsf{tank1}}\langle \ldots e_1, e_2, \ldots \rangle$, where:*

$$\begin{aligned} e_1 &= \langle \mathsf{monitor}, \mathsf{tank1}, \textbf{\textit{set:speed}}(\mathit{new\_speed}) \rangle, \\ e_2 &= \langle \mathit{set\_speed}(55kmh), \mathsf{monitor} \rangle . \end{aligned}$$

5.3 Degrees of Cooperation

There are many different ways in which an agent can make its services secure. One of these ways is to provide no information or to take no action at all, which is a very uncooperative mode of behavior. For example, when the tank1 agent is asked its current speed by the monitor agent, it may choose to protect security by providing no answer at all, even though it is authorized to disclose this information to the monitor agent. Likewise, when the com_c agent is requested by the tank1 agent to provide a safe route to a new location, the com_c agent may respond by merely sending one waypoint to the intended destination instead of a full route, even though it is authorized to disclose a full route. The right balance between security and cooperation depends on a number of application dependent factors.

Independently of exactly what these factors are, there is some notion of *nearness* or *degree of distortion* of an answer or a service. This will be modeled by a partial order on histories as defined below.

Definition 5.8 (More Cooperative History) *For any agent* a*, we use $\leq_{\mathsf{a}}^{\mathsf{coop}}$ to denote a partial order on the set of all histories for* a*. Intuitively, $h \leq_{\mathsf{a}}^{\mathsf{coop}} h'$ means that h' is more cooperative than h.*

Example 5.8 *Consider the following two histories for our* tank1 *agent.*
$h_{\mathsf{tank1}} = \langle \ldots e_1, e_2, \ldots \rangle$, where:

$$\begin{aligned} e_1 &= \langle \mathtt{com\_c}, \mathsf{tank1}, \textbf{\textit{status}}() \rangle, \\ e_2 &= \langle \mathsf{tank1}, \mathtt{com\_c}, \{ \mathbf{in}(\mathtt{low}, \mathsf{tank1} : \textbf{\textit{fuel\_level}}(\mathtt{X}_{now})), \\ &\qquad \mathbf{in}(\mathtt{low}, \mathsf{tank1} : \textbf{\textit{fuel\_level}}(\mathtt{X}_{now})), \\ &\qquad \mathbf{in}((50, 20, 40), \mathsf{tank1} : \textbf{\textit{location}}(\mathtt{X}_{now})) \} \rangle \end{aligned}$$

and

$h'_{\mathsf{tank1}} = \langle \ldots e'_1, e'_2, \ldots \rangle$, where:

$$\begin{aligned} e'_1 &= \langle \mathtt{com\_c}, \mathsf{tank1}, \textbf{\textit{status}}() \rangle \\ e'_2 &= \langle \mathsf{tank1}, \mathtt{com\_c}, \{ \mathbf{in}((50, 20, 40), \mathsf{tank1} : \textbf{\textit{location}}(\mathtt{X}_{now})) \} \rangle . \end{aligned}$$

It seems that in the first history $\mathtt{tank1}$ *is more cooperative by providing more information about its location. Thus, it may be desired to assert that:* $h'_{\mathtt{tank1}} \leq^{\mathsf{coop}}_{\mathtt{tank1}} h_{\mathtt{tank1}}$.

6 Approximating Agent Security

In the preceding section, we have assumed that any agent $\mathtt{b}$ has an associated "true" history, "true" consequence operation, "true" state, etc. However, when an agent $\mathtt{a}$ wants to protect some of its data and/or services from agent $\mathtt{b}$, it needs to know what agent $\mathtt{b}$'s "true" history, consequence operation and state are. In general, this is very difficult to accomplish. Hence, in this section, we introduce the notion of approximations that agent $\mathtt{a}$ may use about another agent $\mathtt{b}$, and we define what it means for such an approximation to be correct w.r.t. the corresponding "true" notion. We show that under appropriate conditions, these approximations guarantee that true data/action security will be preserved. Agent $\mathtt{a}$ does not need to model agent $\mathtt{b}$'s history, consequence operation and state in order to maintain action security. Therefore, in this section we will discuss these approximations in the context of data security.

The organization of this section is as follows.

- First, we define what it means for agent $\mathtt{a}$ to approximate agent $\mathtt{b}$'s history.
- Then, we describe how agent $\mathtt{a}$ approximates agent $\mathtt{b}$'s language (after all, if agent $\mathtt{a}$ knows nothing about agent $\mathtt{b}$'s language, then it cannot say much about agent $\mathtt{b}$'s beliefs).
- Then, we show how these two notions allow us to define how agent $\mathtt{a}$ approximates agent $\mathtt{b}$'s state, given its approximation of agent $\mathtt{b}$'s history and language.
- We then introduce a notion of how agent $\mathtt{a}$ can approximate agent $\mathtt{b}$'s inference mechanism/consequence operation so that it can infer an approximation of agent $\mathtt{b}$'s beliefs.
- Based on these approximations, we show that to preserve security, agent $\mathtt{a}$ must *overestimate* what (it thinks) agent $\mathtt{b}$ will know after it responds to a given request, and it must *underestimate* what (it thinks) agent $\mathtt{b}$ knew before giving the answer.
- Though some of these approximations are space-consuming, we show that all approximations can be *compacted*, but such compactions diminish the level of cooperation agent $\mathtt{a}$ gives to agent $\mathtt{b}$.

6.1 The Basic Idea

The intuition underlying approximate security checks is relatively simple: take the worst possible case and decide what to do on the basis of that worst-case scenario. In our definition of security, we wish to ensure that the set of violated secrets after agent $\mathtt{a}$ provides an answer is a subset of the set of violated secrets before $\mathtt{a}$ gives the answer. Thus, to be safe, we must *underestimate* the set of secrets violated by $\mathtt{b}$ prior to giving an answer, and *overestimate* the set of secrets violated by $\mathtt{b}$ after giving the answer. By underestimating the secrets violated by $\mathtt{b}$ prior to giving an answer, and overestimating the set of secrets violated by $\mathtt{b}$

after giving the answer, we are assuming (as we should in a worst case situation) that the answer causes a maximal set of secrets to be disclosed to the user. The following example illustrates this situation.

Example 6.1 *Consider the scenario described in Example 5.5 in which the* `com_c` *agent may tell the* monitor *agent* tank1*'s location (event* e_0*), and then* tank1 *may tell the* monitor *agent that it is low in fuel (event* e_2*), from which* monitor *can also infer* tank1*'s location.*

Before event e_2*, underestimating the* monitor *agent's set of violated secrets will lead, for example, to not including* tank1*'s location in it. On the other hand, if* monitor *agent's set of violated secrets after* e_2 *is overestimated, it may include* tank1*'s location (inferred from the answer that* tank1 *is low in fuel). These underestimation and overestimation will lead* tank1 *to the conclusion that it cannot tell the* monitor *agent that it is low in fuel. This guarantees data security (but not maximal cooperativeness).*

Summarizing, suppose $\mathtt{a}$ wishes to protect its data from $\mathtt{b}$. Then, in order to perform approximate security checks, $\mathtt{a}$ needs the following items:

- an estimate of $\mathtt{b}$'s possible states;
- an upper bound on the set of secrets that can be derived by $\mathtt{b}$ *using* $\mathtt{a}$*'s answer*;
- a lower bound on the set of secrets that can be derived by $\mathtt{b}$ (from the old state).

In turn, to approximate $\mathtt{b}$'s states, $\mathtt{a}$ needs some approximation of $\mathtt{b}$'s fact language (i.e. of its data structures) and of its history (which influences the actual contents of $\mathtt{b}$'s state). All of these approximate notions are formalized in the succeeding sections.

6.2 Approximating Possible Histories

In this section, we specify what an approximate history is. In order to represent (approximately) $\mathtt{b}$'s history, $\mathtt{a}$ (and its developers) need some language, modeled by the following set.

Definition 6.1 (Possible Histories Approximation) *The set of* approximate history representations for $\mathtt{b}$ used by $\mathtt{a}$ *is a decidable set* $pos\mathcal{H}_{\mathtt{b}}^{\mathtt{a}}$.

In this definition, the set $\mathrm{pos}\mathcal{H}_{\mathtt{b}}^{\mathtt{a}}$ is deliberately generic; there can be many ways to represent $\mathtt{b}$'s histories, and the most appropriate approach will, in general, be application dependent. In particular, the members of $\mathrm{pos}\mathcal{H}_{\mathtt{b}}^{\mathtt{a}}$ may be histories of some sort (as in the following example), or even constraints (as in Section 8) that constitute a partial description of $\mathtt{b}$'s histories. For instance, such constraints may state that $\mathtt{b}$'s history contains a message from $\mathtt{c}$ at some point, and leave the rest of the history unspecified. The need for partial descriptions arises because $\mathtt{a}$ will typically be unable to see all the messages exchanged between $\mathtt{b}$ and other agents. Similarly, $\mathtt{a}$ will be unable to observe all the actions executed by $\mathtt{b}$. An example of an approximate history is given below.

Example 6.2 *Agent* $\mathtt{tank1}$ *may use its own history* $h_{\mathtt{tank1}}$ *as a partial description of the* $\mathtt{monitor}$ *agent's history* $h_{\mathtt{monitor}}$ *. In fact, the messages between* $\mathtt{tank1}$ *and the* $\mathtt{monitor}$ *agent should be the same in* $h_{\mathtt{tank1}}$ *and* $h_{\mathtt{monitor}}$*. Therefore, if* $h_{\mathtt{tank1}} = \langle e_1, e_2, e_3, e_4\rangle$*, where*

$$\begin{aligned}
e_1 &= \langle \mathtt{com\_c}, \mathtt{tank1}, \boldsymbol{set{:}speed}(new\_speed)\rangle,\\
e_2 &= \langle set\_speed(55kmh), \mathtt{com\_c}\rangle,\\
e_3 &= \langle \mathtt{monitor}, \mathtt{tank1}, \boldsymbol{fuel\_level}()\rangle,\\
e_4 &= \langle \mathtt{tank1}, \mathtt{monitor}, \{\mathbf{in}(\mathtt{low}, \mathtt{tank1} : \boldsymbol{fuel\_level}(\mathtt{X}_{now}))\}\rangle .
\end{aligned}$$

then $h_{\mathtt{monitor}}$ *can be any possible history of the form*

$$h_1 \cdot \langle e_3\rangle \cdot h_2 \cdot \langle e_4\rangle \cdot h_3\,, \qquad (1)$$

where h_1*,* h_2 *and* h_3 *contain no message from/to* $\mathtt{tank1}$ *—that is,* $h_{\mathtt{monitor}}$ *can be* any possible history which is $\mathtt{tank1}$-$\mathtt{monitor}$-compatible with $h_{\mathtt{tank1}}$ *(see Definition 5.4).*

The correspondence between approximate and actual histories is application-dependent and, in general, non-trivial. This correspondence is formalized as follows.

Definition 6.2 (History Correspondence Relation $\rightsquigarrow_{\mathsf{h}}$) *For all agents* $\mathtt{a}$ *and* $\mathtt{b}$*, there is an associated* correspondence relation $\rightsquigarrow_{\mathsf{h}} \subseteq pos\mathcal{H}^{\mathtt{a}}_{\mathtt{b}} \times pos\mathcal{H}_{\mathtt{b}}$.

The subscript h will often be omitted to improve readability. Intuitively, if some history $h_{\mathtt{b}} \in \text{pos}\mathcal{H}_{\mathtt{b}}$ matches (under the chosen partial representation of histories) an approximate description $h \in \text{pos}\mathcal{H}^{\mathtt{a}}_{\mathtt{b}}$, then we write $h \rightsquigarrow h_{\mathtt{b}}$. In the above example, $\text{pos}\mathcal{H}^{\mathtt{a}}_{\mathtt{b}}$ coincides with $\text{pos}\mathcal{H}_{\mathtt{a}}$ (the set of all possible histories for $\mathtt{a}$), and $\rightsquigarrow$ coincides with the compatibility relation $\xleftrightarrow{\mathtt{ab}}$. Moreover, agent $\mathtt{a}$ maintains an approximation of $\mathtt{b}$'s current history . This notion is formalized below.

Definition 6.3 (Approximate Current History $AppH_{\mathtt{b}}(.)$, correctness) *Let* $h \in pos\mathcal{H}_{\mathtt{a}}$ *be the current history of* $\mathtt{a}$*. The approximation of* $\mathtt{b}$*'s current history at* h*, is an approximate history representation* $AppH_{\mathtt{b}}(h) \in pos\mathcal{H}^{\mathtt{a}}_{\mathtt{b}}$.

We say that $AppH_{\mathtt{b}}$ *is* correct *if for all* $h \in pos\mathcal{H}_{\mathtt{a}}$*, and for all* $h_{\mathtt{b}} \in pos\mathcal{H}_{\mathtt{b}}$ *such that* $h \xleftrightarrow{ab} h_{\mathtt{b}}$ *it is the case that* $AppH_{\mathtt{b}}(h) \rightsquigarrow h_{\mathtt{b}}$.

Intuitively, an approximate current history is correct if it matches (at least) all possible histories for $\mathtt{b}$ that are compatible with $\mathtt{a}$'s history. An example of a correct approximate history is given below.

Example 6.3 *Suppose, as in Example 6.2, that* $\mathtt{tank1}$ *uses its own histories as a partial description of* $\mathtt{monitor}$*'s possible histories. That is,* $pos\mathcal{H}^{\mathtt{tank1}}_{\mathtt{monitor}}$ *includes the projection of the histories in* $pos\mathcal{H}_{\mathtt{tank1}}$ *on the interactions with agent* $\mathtt{monitor}$*. In addition,* $\rightsquigarrow$ *coincides with the compatibility relation* $\xleftrightarrow{\mathtt{tank1}\,\mathtt{monitor}}$*, and for* $h \in pos\mathcal{H}_{\mathtt{tank1}}$*,* $AppH_{\mathtt{monitor}}(h)$ *is the projection of* h *to the interactions with* $\mathtt{monitor}$*. Then,* $AppH_{\mathtt{monitor}}$ *is correct.*

6.3 Approximating Languages

The first difficulty in approximating $\mathfrak{b}$'s state is that $\mathfrak{a}$ may have imprecise knowledge of $\mathfrak{b}$'s fact language (i.e. of the data structures and function calls used by $\mathfrak{b}$). $\mathfrak{a}$ is forced to use some ground code calls, and hope that these code calls mimic the operations that $\mathfrak{b}$ actually has in its repertoire.[2]

Definition 6.4 (Approximate Fact Language $App\mathcal{L}_{\mathfrak{b}}$**)** *The approximate fact language of* $\mathfrak{b}$ *used by* $\mathfrak{a}$ *is a denumerable set* $App\mathcal{L}_{\mathfrak{b}}$ *.*

The relationship between the approximate fact language used by $\mathfrak{a}$ and the actual fact language used by $\mathfrak{b}$ is formalized by the following *fact correspondence relation*, that relates approximate facts to the actual data structures of $\mathfrak{b}$ that match the approximate description.

Definition 6.5 (Fact Correspondence Relation $\leadsto_{\mathsf{f}}$**)** *For all agents* $\mathfrak{a}$ *and* $\mathfrak{b}$*, there is an associated* fact correspondence relation $\leadsto_{\mathsf{f}} \subseteq App\mathcal{L}_{\mathfrak{b}} \times \mathcal{L}_{\mathfrak{b}}$ *.*

We drop the subscript f whenever the context allows us to distinguish $\leadsto_{\mathsf{h}}$ from $\leadsto_{\mathsf{f}}$

Intuitively, we write $f \leadsto f_{\mathfrak{b}}$ if $f_{\mathfrak{b}}$ is one of the possible instantiated data structures for $\mathfrak{b}$ that match the approximate description f used by $\mathfrak{a}$.

Some approximate facts f may have no counterpart in $\mathcal{L}_{\mathfrak{b}}$ (e.g. $\mathfrak{a}$ may think that $\mathfrak{b}$ can use a code call $\mathtt{p} : g()$ when in fact this is not the case). In such cases, we write:

$$f \not\leadsto \quad \text{if, by definition,} \quad \nexists f'.\ f \leadsto f' .$$

Analogously, some facts of $\mathcal{L}_{\mathfrak{b}}$ may have no approximate counterpart (e.g. when $\mathfrak{a}$ does not know that $\mathfrak{b}$ may use some code call $\mathtt{p} : h()$). In this case we write:

$$\not\leadsto f \quad \text{if, by definition,} \quad \nexists f'.\ f' \leadsto f .$$

Ground code call conditions are approximated by sets of approximate facts. Approximate conditions are matched against sets of facts from $\mathcal{L}_{\mathfrak{b}}$ by means of a correspondence relation derived from the correspondence relation for individual facts, $\leadsto_{\mathsf{f}}$.

Definition 6.6 (Approximate Conditions) *An approximate condition is a set* $C \subseteq App\mathcal{L}_{\mathfrak{b}}$ *.*

Definition 6.7 (Condition Correspondence Relation) *We say that an approximate condition* $C \subseteq App\mathcal{L}_{\mathfrak{b}}$ *corresponds to a set of facts* $C_{\mathfrak{b}} \subseteq \mathcal{L}_{\mathfrak{b}}$*, denoted* $C \leadsto_{\mathsf{c}} C_{\mathfrak{b}}$*, if both the following conditions hold:*

1. *if* $f \in C$ *then either* $f \not\leadsto$ *or* $\exists f_{\mathfrak{b}} \in C_{\mathfrak{b}}\ .\ f \leadsto f_{\mathfrak{b}}$ *.*
2. *if* $f_{\mathfrak{b}} \in C_{\mathfrak{b}}$ *then either* $\not\leadsto f_{\mathfrak{b}}$ *or* $\exists f \in C\ .\ f \leadsto f_{\mathfrak{b}}$ *.*

[2] In the following definitions, when the approximating agent, $\mathfrak{a}$, is clear from the context, we will omit it from the notation. For example, we will write $App\mathcal{L}_{\mathfrak{b}}$ instead of $App\mathcal{L}_{\mathfrak{b}}^{\mathfrak{a}}$.

The first requirement above says that all elements, f, of the approximate condition must correspond to some fact f_{b} in the actual state unless f has no counterpart in the language $\mathcal{L}_{\mathrm{b}}$ (in which case, f is ignored). Similarly, the second requirement says that each member of C_{b} must have a counterpart in C, with the exception of those facts f_{b} that are not expressible in the approximate language $App\mathcal{L}_{\mathrm{b}}$. The following example describes how states are approximated in the Tank example.

Example 6.4 *Suppose the code calls in* $\mathcal{L}_{\mathrm{monitor}}$ *include:*

$$\begin{array}{rl} \mathbf{in}(\mathtt{P}, \mathrm{tank1} : \mathit{location}()), & \mathbf{in}(\mathtt{F}, \mathrm{tank1} : \mathit{fuel\_level}()), \\ \mathbf{in}(\mathtt{S}, \mathrm{tank1} : \mathit{soldiers}()), & \mathbf{in}(\mathtt{D}, \mathrm{monitor} : \mathit{distance}(\mathtt{A1}, \mathtt{A2})), \\ \mathbf{in}(\mathtt{Y}, \mathrm{monitor} : \mathit{repair\_needed}(\mathtt{A})) . & \end{array}$$

Agent tank1 *may think that the functions* **location** *and* **fuel_level** *used by* monitor *have one argument, e.g., time* T. *It may not know that* $\mathbf{in}(\mathtt{D}, \mathrm{monitor} : \mathit{distance}(\mathtt{A1}, \mathtt{A2}))$ *is used by the* monitor *agent and may think that it uses* $\mathit{status}(\mathtt{A})$ *instead of* $\mathit{repair\_needed}(\mathtt{A})$. *In addition,* tank1 *may think that the* monitor *also uses* $\mathbf{in}(\mathtt{R}, \mathrm{monitor} : \mathit{region}(\mathtt{T}))$. *Thus,* $App\mathcal{L}^{\mathrm{tank1}}_{\mathrm{monitor}}$ *may include:*

$$\begin{array}{rl} \mathbf{in}(\mathtt{P}, \mathrm{tank1} : \mathit{location}(\mathtt{T})), & \mathbf{in}(\mathtt{L}, \mathrm{tank1} : \mathit{fuel\_level}(\mathtt{T})), \\ \mathbf{in}(\mathtt{S}, \mathrm{tank1} : \mathit{soldiers}()), & \mathbf{in}(\mathtt{Z}, \mathrm{tank1} : \mathit{region}(\mathtt{T})), \\ \mathbf{in}(\mathtt{Y}, \mathrm{monitor} : \mathit{status}(\mathtt{A})) . & \end{array}$$

where, for example,

$$\begin{array}{rcl} \mathbf{in}(\mathtt{Y}, \mathrm{monitor} : \mathit{status}(\mathtt{A})) & \leadsto_{\mathsf{f}} & \mathbf{in}(\mathtt{Y}, \mathrm{monitor} : \mathit{repair\_needed}(\mathtt{A})), \\ \mathbf{in}(\mathtt{Z}, \mathit{region} : \mathtt{T}()) & \not\leadsto_{\mathsf{f}} & \\ & \not\leadsto_{\mathsf{f}} & \mathbf{in}(\mathtt{D}, \mathrm{monitor} : \mathit{distance}(\mathtt{A1}, \mathtt{A2})) . \end{array}$$

For example, if the condition
$\{\mathbf{in}(\mathbf{true}, \mathrm{monitor} : \mathit{repair\_needed}(\mathrm{tank1})), \mathbf{in}(\mathtt{north\_east}, \mathrm{tank1} : \mathit{region}(\mathtt{X}_{now}))\}$ *is in* monitor*'s approximation, it may correspond to*
$\{\mathbf{in}(\mathtt{need\_repair}, \mathrm{monitor} : \mathit{status}(\mathrm{tank1})), \mathbf{in}(5, \mathrm{monitor} : \mathit{distance}(\mathrm{tank1}, \mathrm{monitor}))\}$.

6.4 Approximating States

The approximation of a state $\mathcal{O}_{\mathrm{b}}$ should tell us the following things:

- which facts are *surely true* in $\mathcal{O}_{\mathrm{b}}$; this is needed by a to underestimate the inferences of b (inferences can be part of a correct underestimation only if they follow from conditions that are guaranteed to be true in $\mathcal{O}_{\mathrm{b}}$);
- which facts *may possibly be true* in $\mathcal{O}_{\mathrm{b}}$; this is needed by a to overestimate the inferences of b (inferences that depend on facts that *might* be in $\mathcal{O}_{\mathrm{b}}$ should be considered by the overestimation);

- which facts are *new*; this is needed to identify the inferences that really depend on the last answer; intuitively, a new secret is violated only when it is derived from some new fact.

Accordingly, approximate states are described using three sets of approximate conditions.

Definition 6.8 (Approximate States $App\mathcal{O}_{\mathfrak{b}} = \langle Nec, Poss, New\rangle$**)** *An approximate state of* $\mathfrak{b}$ *used by* $\mathfrak{a}$ *is a triple* $App\mathcal{O}_{\mathfrak{b}} = \langle Nec, Poss, New\rangle$*, whose elements are sets of approximate conditions (i.e.* $App\mathcal{O}_{\mathfrak{b}} \in \wp(App\mathcal{L}_{\mathfrak{b}}) \times \wp(App\mathcal{L}_{\mathfrak{b}}) \times \wp(App\mathcal{L}_{\mathfrak{b}})$ *). The three elements of an approximate state* $App\mathcal{O}_{\mathfrak{b}}$ *will be denoted by* $App\mathcal{O}_{\mathfrak{b}}.Nec$*,* $App\mathcal{O}_{\mathfrak{b}}.Poss$*, and* $App\mathcal{O}_{\mathfrak{b}}.New$*, respectively.* $App\mathcal{O}_{\mathfrak{b}}$ *is required to satisfy the following inclusions:*

1. $App\mathcal{O}_{\mathfrak{b}}.Nec \subseteq App\mathcal{O}_{\mathfrak{b}}.Poss$ *;*
2. $App\mathcal{O}_{\mathfrak{b}}.New \subseteq App\mathcal{O}_{\mathfrak{b}}.Poss$ *.*

The first inclusion says that a condition C cannot be necessarily true if it is not possibly true. The second inclusion says that all new facts must be possible.

Agent $\mathfrak{a}$ maintains an approximation of $\mathfrak{b}$'s current state. This is formalized via the following definition.

Definition 6.9 (Approximate State Function $App\mathcal{O}_{\mathfrak{b}}$**, correctness)** *The approximate state function* $App\mathcal{O}_{\mathfrak{b}}$ *is a mapping which maps approximate histories from* $pos\mathcal{H}^{\mathfrak{a}}_{\mathfrak{b}}$ *onto approximate states of* $\mathfrak{b}$ *used by* $\mathfrak{a}$*. We say that* $App\mathcal{O}_{\mathfrak{b}}$ *is* correct *if for all approximate histories* $h \in pos\mathcal{H}^{\mathfrak{a}}_{\mathfrak{b}}$*, the following conditions hold:*

1. *if* $C \in App\mathcal{O}_{\mathfrak{b}}(h).Nec$*, then for all* $h_{\mathfrak{b}}$ *such that* $h \rightsquigarrow h_{\mathfrak{b}}$ *there exists* $C_{\mathfrak{b}} \subseteq \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}})$ *such that* $C \rightsquigarrow C_{\mathfrak{b}}$ *;*
2. *for all* $C_{\mathfrak{b}} \subseteq \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}})$ *such that* $h \rightsquigarrow h_{\mathfrak{b}}$*, if* $C \rightsquigarrow C_{\mathfrak{b}}$ *then* $C \in App\mathcal{O}_{\mathfrak{b}}(h).Poss$ *;*
3. *for all possible non-empty histories* $h_{\mathfrak{b}} \cdot e \in pos\mathcal{H}_{\mathfrak{b}}$ *such that* $h \rightsquigarrow h_{\mathfrak{b}} \cdot e$*, and for all* $C_{\mathfrak{b}} \subseteq \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}} \cdot e)$ *such that* $C_{\mathfrak{b}} \not\subseteq \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}})$*, if* $C \rightsquigarrow C_{\mathfrak{b}}$ *then* $C \in App\mathcal{O}_{\mathfrak{b}}(h).New$ *.*

Intuitively, the above correctness conditions state that: (i) each condition C in *Nec* should correspond to some condition $C_{\mathfrak{b}}$ which is actually true in the current state of $\mathfrak{b}$, *whatever it may be* (note the universal quantification over $h_{\mathfrak{b}}$); thus, in case of doubt, in order to achieve correctness it is better to underestimate *Nec*; (ii) the approximations C of each set of facts $C_{\mathfrak{b}}$ that might be part of $\mathfrak{b}$'s current state should be included in *Poss* (in case of doubt, it is better to overestimate *Poss* to achieve correctness); (iii) if a set of facts is new in $\mathfrak{b}$'s current state (because $C_{\mathfrak{b}} \subseteq \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}} \cdot e)$ and $C_{\mathfrak{b}} \not\subseteq \mathcal{O}_{\mathfrak{b}}(h_{\mathfrak{b}})$), then its counterparts C should be included in *New* (that should be overestimated in case of doubt). An example of an approximate state function that is correct is shown below.

Example 6.5 *Consider the scenario depicted in Example 6.4. Suppose the approximate language* $App\mathcal{L}_{\mathsf{monitor}}$ *contains only the code calls*
$\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T}))$, $\mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))$ *and* $\mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())$ *where* $\mathtt{P}$ *is* $\mathsf{tank1}$*'s current position at time* $\mathtt{T}$ *and* $\mathtt{S}$ *is the list of* $\mathsf{tank1}$*'s soldiers. Suppose* h_{tank1}

is a history of $\mathsf{tank1}$ *in which it did not send any answers to the* $\mathsf{monitor}$ *agent. Consider the scenario in which the* $\mathsf{monitor}$ *agent didn't receive any answers from other agents (including* $\mathsf{tank1}$*) and* $\mathsf{tank1}$ *also believes it. That is, in* $AppH_{\mathsf{monitor}}(h_{\mathsf{tank1}}) = h$ *there is no answers sent to the* $\mathsf{monitor}$ *agent from* $\mathsf{tank1}$*, In this case one should set:*

$$\begin{array}{lcl}
AppO_{\mathsf{monitor}}(h).Nec & = & \emptyset\,, \\
AppO_{\mathsf{monitor}}(h).Poss & = & \{\{\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T}))\}, \\
& & \ \{\mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
& & \ \{\mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))\}\,, \\
& & \ \{\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T}))\,,\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
& & \ \{\mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T})), \mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T}))\}\,, \\
& & \ \{\mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T})), \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}\,, \\
& & \ \{\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T}))\,,\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}()), \\
& & \quad \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))\} \\
\}\,. & &
\end{array}$$

In other words, nothing is necessary, everything is possible. If $\mathsf{tank1}$ *sent the* $\mathsf{monitor}$ *agent an answer message* $e = \langle \mathsf{tank1}, \mathsf{monitor}, \{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\}\rangle$*, then one might set:*

$$\begin{array}{lcl}
AppO_{\mathsf{monitor}}(h \cdot e).Nec & = & \{\{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\}\}\,, \\
AppO_{\mathsf{monitor}}(h \cdot e).Poss & = & AppO_{\mathsf{monitor}}(h).Poss \\
AppO_{\mathsf{monitor}}(h \cdot e).New & = & \{\{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\}, \\
& & \ \{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\,,\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
& & \ \{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\,,\ \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))\}, \\
& & \ \{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\,,\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
& & \quad \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))\}\}\,.
\end{array}$$

Note that in this example $\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))$ *becomes necessarily true (in some other cases,* $\mathsf{monitor}$ *might disbelieve* $\mathsf{tank1}$*, and* $AppO_{\mathsf{monitor}}(h \cdot e).Nec$ *would remain empty). The set of possible new conditions that become true due to e is set to all the sets of facts that contain the answer* $\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))$*. Only secrets that are revealed from new facts are due to security violation of* $\mathsf{tank1}$*.*

Consider a third variation of this scenario where the $\mathsf{com\_c}$ *agent has told the* $\mathsf{monitor}$ *agent the list of soldiers of* $\mathsf{tank1}$ *and suppose that* $\mathsf{tank1}$ *believes that this happened (as approximated by* h'*) and that the* $\mathsf{monitor}$ *agent does not forget such lists. Assume further, that* $e' = \langle \mathsf{tank1}, \mathsf{monitor}, \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\rangle$*. In this case*

$$\begin{array}{lcl}
AppO_{\mathsf{monitor}}(h').Nec & = & \{\{\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
AppO_{\mathsf{monitor}}(h' \cdot e').New & = & \{\{\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
& & \ \{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\,,\ \mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}())\}, \\
& & \ \{\mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}()), \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))\}, \\
& & \ \{\mathbf{in}(\mathtt{S}, \mathsf{tank1} : \mathit{soldiers}()), \mathbf{in}(\mathtt{low}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now})), \\
& & \quad \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \mathit{location}(\mathtt{T}))\} \\
\}\,. & &
\end{array}$$

6.5 Approximate Secrets

In the framework of exact security checks, when agent a describes the set of secrets $Sec_{\mathsf{a}}(\mathsf{b})$ it wishes to prevent b from inferring, the members of $Sec_{\mathsf{a}}(\mathsf{b})$ are drawn from $\mathcal{L}_{\mathsf{b}}$. As this language itself may only be partially known to agent $\mathfrak{a}$, $\mathfrak{a}$ must use some approximation of its secrets function.

Definition 6.10 (Approximate Secrets $AppSec(\mathfrak{b})$) *The set of* approximate secrets *of agent $\mathfrak{a}$ w.r.t. agent $\mathfrak{b}$, denoted by $AppSec(\mathfrak{b})$, is some subset of $App\mathcal{L}_{\mathfrak{b}}$.*

Clearly, the fact correspondence relation $\rightsquigarrow_{\mathfrak{f}}$ applies to approximate secrets. If $f \in AppSec(\mathfrak{b})$ approximates $f' \in Sec_{\mathfrak{a}}(\mathfrak{b})$, then we write $f \rightsquigarrow f'$. What it means for a set of approximate secrets to be correct is defined below.

Definition 6.11 (Approximate Secrets, Correctness) *The set $AppSec(\mathfrak{b})$ is correct w.r.t. $Sec_{\mathfrak{a}}(\mathfrak{b})$ if it satisfies the following conditions:*

1. *for all $f_{\mathfrak{b}} \in Sec_{\mathfrak{a}}(\mathfrak{b})$ there exists $f \in AppSec(\mathfrak{b})$ such that $f \rightsquigarrow f_{\mathfrak{b}}$;*
2. *if $f \rightsquigarrow f_{\mathfrak{b}}$ and $f_{\mathfrak{b}} \in Sec_{\mathfrak{a}}(\mathfrak{b})$, then $f \in AppSec(\mathfrak{b})$.*

Condition 1 says that each secret should be expressible in the approximate language $App\mathcal{L}_{\mathfrak{b}}$ (otherwise, some violation might go unnoticed). Condition (2) above states the conservative principle that if a fact f may correspond to a secret, then it should be treated like a secret.

6.6 Approximate Consequences

In this section, we define what it means for an agent $\mathfrak{a}$ to correctly approximate agent $\mathfrak{b}$'s consequence operation. We start by defining the type of the approximation.

Definition 6.12 (Approximate Consequence Operation) *An approximate consequence operation of $\mathfrak{b}$ used by $\mathfrak{a}$ is a mapping of type $\wp(App\mathcal{L}_{\mathfrak{b}}) \to \wp(App\mathcal{L}_{\mathfrak{b}})$.*

Recall that when providing an answer to agent $\mathfrak{b}$, agent $\mathfrak{a}$ should underestimate what is known to $\mathfrak{b}$ prior to providing the answer, and should overestimate what is known to $\mathfrak{b}$ after providing the answer. This may be done by using approximate consequence functions that underestimate and overestimate $\mathfrak{b}$'s actual consequence function.

Definition 6.13 (Correct Underestimate) *An approximate consequence operation $UCn_{\mathfrak{b}}$ is a correct underestimate of $Cn_{\mathfrak{b}}$ if, for all abstract conditions C and abstract facts f, if $f \in UCn_{\mathfrak{b}}(C)$ then for all $C_{\mathfrak{b}}$ and $f_{\mathfrak{b}}$ such that $C \rightsquigarrow C_{\mathfrak{b}}$ and $f \rightsquigarrow f_{\mathfrak{b}}$, it holds that $C_{\mathfrak{b}} \vdash_{\mathfrak{b}} f_{\mathfrak{b}}$.*

In other words, $UCn_{\mathfrak{b}}$ is a correct underestimate of $\mathrm{Cn}_{\mathfrak{b}}$ if what can be inferred using $UCn_{\mathfrak{b}}$ is also derivable using $\vdash_{\mathfrak{b}}$ (and hence $\mathrm{Cn}_{\mathfrak{b}}$). Here we use $\vdash_{\mathfrak{b}}$ instead of $\mathrm{Cn}_{\mathfrak{b}}$ because $C_{\mathfrak{b}}$ is only a *partial* description of the contents of $\mathfrak{b}$'s state (cf. the discussion in Section 4.3).

The following example provides a correct underestimate in the case of the Tank example.

Example 6.6 *Consider the tank example. The identity function is a correct underestimate of the* $\mathtt{monitor}$ *agent's consequence operation. That is, $\forall C \subseteq App\mathcal{L}_{\mathtt{monitor}}$, $UCn^{\mathtt{tank1}}_{\mathtt{monitor}}(C) = C$*

Before proceeding to the definition of correct overestimates, we need a definition that intuitively captures the *causal dependencies* between a set $C_{\mathtt{b}}$ of facts and the facts $f_{\mathtt{b}}$ that can be derived from $C_{\mathtt{b}}$. This is needed to focus on the secrets that are violated *because* of $\mathtt{a}$'s answer as demonstrated in the following example.

Example 6.7 *Consider the scenario of Example 6.5 and suppose that* $\mathtt{tank1}$ *would like to protect its soldiers list from the* $\mathtt{monitor}$ *agent. It is clear that giving the answer on its fuel level, as in event e, has nothing to do with the soldiers list. However, the set*

$$\{\mathbf{in}(\mathtt{low}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))\ ,\ \mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{soldiers}())\}$$

that is in $AppO_{\mathtt{monitor}}(h \cdot e).New$ *does entail* $\mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{soldiers}())$*, i.e., the secret. Including the answer* $\mathbf{in}(\mathtt{low}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{X}_{now}))$ *in every set of New does not help. For this, we will need the notion of "causality."*

The mapping $\mathrm{Cn}_{\mathtt{b}}$ is not completely adequate for defining and overestimating the consequence operation because in general, when $f_{\mathtt{b}} \in \mathrm{Cn}_{\mathtt{b}}(C_{\mathtt{b}})$, $C_{\mathtt{b}}$ may contain facts that are not relevant to the proof of $f_{\mathtt{b}}$. Rather, we should say that $f_{\mathtt{b}}$ is caused by the presence of $C_{\mathtt{b}}$ when $f_{\mathtt{b}} \in \mathrm{Cn}_{\mathtt{b}}(C_{\mathtt{b}})$ *and* $C_{\mathtt{b}}$ is *minimal*, i.e. if we dropped even one fact from $C_{\mathtt{b}}$, then $f_{\mathtt{b}}$ would not be derivable anymore.

Definition 6.14 (Causal Dependencies) *We say that* $C_{\mathtt{b}}$ *causes* $f_{\mathtt{b}}$*, denoted* $\mathsf{Causes}(C_{\mathtt{b}}, f_{\mathtt{b}})$*, if* $C_{\mathtt{b}} \vdash_{\mathtt{b}} f_{\mathtt{b}}$ *and for all* $C \subset C_{\mathtt{b}}$*,* $C \not\vdash_{\mathtt{b}} f_{\mathtt{b}}$ *.*

We are now ready to give a formal definition of correct overestimates. From the standpoint of security, it is not necessary that a correct overestimate of $\mathtt{b}$'s consequence operation contain all inferences that $\mathtt{b}$ can draw. Rather, we only require that the overestimate include all possible secrets that $\mathtt{b}$ may infer. This is captured by the following definition.

Definition 6.15 (Correct Overestimate) *An approximate consequence operation* $OCn_{\mathtt{b}}$ *is a correct overestimate of* $Cn_{\mathtt{b}}$ *if for all* $C_{\mathtt{b}}$ *and* $f_{\mathtt{b}}$ *such that* $C_{\mathtt{b}}$ *causes* $f_{\mathtt{b}}$ *and* $f_{\mathtt{b}} \in Sec_{\mathtt{a}}(\mathtt{b})$*, there exist* C, f *such that* $C \rightsquigarrow C_{\mathtt{b}}$ *,* $f \rightsquigarrow f_{\mathtt{b}}$ *, and* $f \in OCn_{\mathtt{b}}(C)$ *.*

The following example shows a simple correct overestimate in the context of the Tank Example.

Example 6.8 *We return to Example 6.5 and assume that the* $\mathtt{monitor}$ *agent can sometimes infer* $\mathtt{tank1}$*'s location from its being low in fuel, and otherwise has the identity consequence operation.*

If the only consequences that $\mathtt{tank1}$ *includes in its overestimation of* $\mathtt{monitor}$*'s consequence operation are the following, then it is a correct overestimation.*

$$\begin{aligned}
OCn^{\mathtt{tank1}}_{\mathtt{monitor}}(\{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{soldiers}())\}) &= \{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{soldiers}())\}, \\
OCn^{\mathtt{tank1}}_{\mathtt{monitor}}(\{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{location}(\mathtt{T}))\}) &= \{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{location}(\mathtt{T}))\}, \\
OCn^{\mathtt{tank1}}_{\mathtt{monitor}}(\{\mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T}))\}) &= \{\mathbf{in}(\mathtt{P}, \mathtt{tank1} : \mathit{location}(\mathtt{T})), \\
&\quad\ \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T}))\}\,.
\end{aligned}$$

The overestimation in general can be a strict subset of all the facts that can be handled by $\mathtt{monitor}$*, for example,*

$$\mathbf{in}(\mathtt{S}, \mathtt{tank1} : \mathit{soldiers}()) \notin OCn^{\mathtt{tank1}}_{\mathtt{monitor}}(\mathbf{in}(\mathtt{L}, \mathtt{tank1} : \boldsymbol{\mathit{fuel\_level}}(\mathtt{T}))\}) \,.$$

6.7 Approximate Data Security Check

In this section, we have defined what it means for an approximate history to be correct, an approximate consequence operation to be a correct under/over estimate of another agent's consequence operation, etc. In short, agent $\mathtt{a}$ approximates $\mathtt{b}$'s behavior via the functions $AppH_{\mathtt{b}}$, $App\mathcal{O}_{\mathtt{b}}$, $OCn_{\mathtt{b}}$ and $UCn_{\mathtt{b}}$. The secrets in $Sec_{\mathtt{a}}(\mathtt{b})$ are approximated by $AppSec(\mathtt{b})$. Together, these functions constitute $\mathtt{a}$'s approximate view of $\mathtt{b}$.

Definition 6.16 (Agent Approximation, correctness) *The approximation of* $\mathtt{b}$ *used by* $\mathtt{a}$ *(based on the approximate languages* $pos\mathcal{H}^{\mathtt{a}}_{\mathtt{b}}$ *and* $App\mathcal{L}_{\mathtt{b}}$*, and on the correspondence functions* $\rightsquigarrow_{\mathsf{h}}$ *and* $\rightsquigarrow_{\mathsf{f}}$*) is a quintuple*

$$\mathbf{App}(\mathtt{b}) = \langle AppH_{\mathtt{b}},\ App\mathcal{O}_{\mathtt{b}},\ AppSec(\mathtt{b}),\ OCn_{\mathtt{b}},\ UCn_{\mathtt{b}} \rangle,$$

whose members are, respectively, a current history approximation , a current state approximation, a set of approximate secrets and two approximate consequence operations.

We say that $\mathbf{App}(\mathtt{b})$ *is* correct *if* $AppH_{\mathtt{b}}$*,* $App\mathcal{O}_{\mathtt{b}}$ *and* $AppSec(\mathtt{b})$ *are correct,* $OCn_{\mathtt{b}}$ *is a correct overestimate of* $Cn_{\mathtt{b}}$*, and* $UCn_{\mathtt{b}}$ *is a correct underestimate of* $Cn_{\mathtt{b}}$*.*

This definition builds upon definitions of what it means for the individual components of $\mathbf{App}(\mathtt{b})$ to be correct—something we have defined in preceding sections of this paper.

Using these concepts, we wish to specify what it means for a history to be approximately data secure. If we can compute an overestimate of the set of secrets violated by agent $\mathtt{b}$ *after* agent $\mathtt{a}$ provides an answer to its request, and we compute an underestimate of the set of secrets violated by agent $\mathtt{b}$ *before* agent $\mathtt{a}$ provides an answer, and if we can show that the latter is a superset of the former, then we would be able to safely guarantee data security. We first define these over/under estimates below, and then use those definitions to define what it means for a history to be approximately data secure.

Definition 6.17 (Overestimate of Violated Secrets) *For all approximate histories* $h \in pos\mathcal{H}^{\mathtt{a}}_{\mathtt{b}}$ *let*

$$\mathsf{OViol}_{\mathtt{b}}(h) =_{def} \bigcup \{ OCn_{\mathtt{b}}(C) \mid C \in App\mathcal{O}_{\mathtt{b}}(h).New\} \cap AppSec(\mathtt{b}) \,.$$

Informally, $\mathsf{OViol}_{\mathtt{b}}(h)$ is the overestimated set of secrets that can be derived because of some new facts (the reason why only the consequences of new facts are considered is illustrated earlier via Example 6.7.)

Definition 6.18 (Underestimate of Violated Secrets) *For all approximate histories* $h \in pos\mathcal{H}^{\mathtt{a}}_{\mathtt{b}}$*, let*

$$\mathsf{UViol}_{\mathtt{b}}(h) =_{def} \bigcup \{ UCn_{\mathtt{b}}(C) \mid C \in App\mathcal{O}_{\mathtt{b}}(h).Nec\} \cap AppSec(\mathtt{b}) \,.$$

In other words, $\mathsf{UViol}_{\mathtt{b}}(h)$ is the underestimated set of secrets that can be derived from facts which are estimated to be necessarily true. The following example illustrates the notions of over/underestimates of violated secrets.

Example 6.9 *We return to Example 6.5 and assume that $UCn^{\mathtt{tank1}}_{\mathtt{monitor}}$ is the identity function and $OCn^{\mathtt{tank1}}_{\mathtt{monitor}}$ is as defined in Example 6.8 and that* $\mathtt{tank1}$ *would like to protect its current location and its soldiers list.*

Consider the second scenario of Example 6.5 where there is no interaction between the $\mathtt{monitor}$ *agent and the other agents in h. As $App\mathcal{O}_{\mathtt{monitor}}(h).Nec$ is empty, $\mathsf{UViol}_{\mathtt{monitor}}(h)$ is also empty. However, $\mathsf{OViol}_{\mathtt{monitor}}(h \cdot e) = \{\mathbf{in}((50, 20, 40), \mathtt{tank1} : \boldsymbol{location}(\mathtt{X}_{now}))\}$ because one of the sets in New causes it and it is a secret.*

In the third scenario of Example 6.5, $\mathsf{UViol}_{\mathtt{monitor}}(h') = \{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : soldiers())\}$ and in addition, $\mathsf{OViol}_{\mathtt{monitor}}(h' \cdot e') = \{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : soldiers())\}$.

We may now define the approximate counterpart of data security.

Definition 6.19 (Approximate Data Security) *A history $h_{\mathtt{a}} \in pos\mathcal{H}_{\mathtt{a}}$ is* approximately data secure w.r.t. $\mathbf{App}(\mathtt{b})$ *if for all initial segments $h' \cdot e$ of $h_{\mathtt{a}}$ such that e is an answer message $\langle \mathtt{a}, \mathtt{b}, Ans\rangle$,*

$$\mathsf{UViol}_{\mathtt{b}}(AppH_{\mathtt{b}}(h')) \supseteq \mathsf{OViol}_{\mathtt{b}}(AppH_{\mathtt{b}}(h' \cdot e))\,.$$

If all histories $h_{\mathtt{a}} \in pos\mathcal{H}_{\mathtt{a}}$ are approximately data secure w.r.t. $\mathbf{App}(\mathtt{b})$, then we say that $\mathtt{a}$ is approximately data secure w.r.t. $\mathbf{App}(\mathtt{b})$.

We reiterate that we are comparing an overestimate of the secrets violated by $\mathtt{b}$ due to $\mathtt{a}$'s answer e (right-hand side of the above inclusion), with an underestimate of the secrets violated by $\mathtt{b}$ before the answer (left-hand side of the inclusion). The following example shows an approximately data secure history.

Example 6.10 *In the second scenario specified in Examples 6.5 and 6.9, it is clear that $h_{\mathtt{tank1}} \cdot e$ is not approximately data secure w.r.t the approximations we described in the previous examples as $AppH_{\mathtt{monitor}}(h_{\mathtt{tank1}}) = h$, $\mathsf{UViol}_{\mathtt{monitor}}(h)$ is empty and when e is the event in which* $\mathtt{tank1}$ *tells the* $\mathtt{monitor}$ *agent that it is low in fuel,*

$$\mathsf{OViol}_{\mathtt{monitor}}(h \cdot e) = \{\mathbf{in}((50, 20, 40), \mathtt{tank1} : \boldsymbol{location}(\mathtt{X}_{now}))\}\,.$$

Thus, $\mathsf{UViol}_{\mathtt{monitor}}(h) \not\supseteq \mathsf{OViol}_{\mathtt{monitor}}(h \cdot e)$.

However, when $AppH_{\mathtt{monitor}}(h_{\mathtt{tank1}}) = h'$ in which the $\mathtt{monitor}$ *agent received the soldiers list from* $\mathtt{com\_c}$*, and e' is the event in which* $\mathtt{tank1}$ *gives the* $\mathtt{monitor}$ *agent its soldiers list, then $h_{\mathtt{tank1}} \cdot e'$ is approximately data secure, while $h_{\mathtt{tank1}} \cdot e$ is not.*

The approximate data security check works well if the approximation $\mathbf{App}(\mathtt{b})$ is *correct*. The theorem below shows that, under this assumption, the approximate security check correctly enforces the "true" notion of data security. As a consequence, if the designer of agent $\mathtt{a}$ can ensure that the approximation of agent $\mathtt{b}$ is correct, then "true" data security is guaranteed by the approximation, even though the agent $\mathtt{a}$ doesn't precisely know the history, state, consequence operation, etc. used by agent $\mathtt{b}$.

Theorem 6.1 (Correct Approximate Data Security Implies Data Security) *If $h_\mathfrak{a}$ is approximately data secure w.r.t.* **App**$(\mathfrak{b})$ *and* **App**$(\mathfrak{b})$ *is correct, then $h_\mathfrak{a}$ is data secure w.r.t.* $\mathfrak{b}$*.*

Proof: We prove the contrapositive, which is equivalent. Suppose $h_\mathfrak{a}$ is *not* data secure w.r.t. $\mathfrak{b}$. Then, for some prefix $h' \cdot e$ of $h_\mathfrak{a}$, where $e = \langle \mathfrak{a}, \mathfrak{b}, Ans \rangle$, and for some history $h_\mathfrak{b} \cdot e \in \mathrm{pos}\mathcal{H}_\mathfrak{b}$, it holds that $h_\mathfrak{b} \cdot e \overset{\mathfrak{ab}}{\Longleftrightarrow} h' \cdot e$ and

$$\mathsf{Violated}_\mathfrak{b}^\mathfrak{a}(h_\mathfrak{b}) \not\supseteq \mathsf{Violated}_\mathfrak{b}^\mathfrak{a}(h_\mathfrak{b} \cdot e)\,.$$

Consequently, there exists $f_0 \in Sec_\mathfrak{a}(\mathfrak{b})$ such that

(a) $f_0 \in \mathsf{Violated}_\mathfrak{b}^\mathfrak{a}(h_\mathfrak{b} \cdot e)$ and

(b) $f_0 \notin \mathsf{Violated}_\mathfrak{b}^\mathfrak{a}(h_\mathfrak{b})$.

Claim 1: there exists f_1 such that $f_1 \rightsquigarrow f_0$ and $f_1 \in \mathsf{OViol}_\mathfrak{b}(AppH_\mathfrak{b}(h' \cdot e))$.

This claim can be proved via the following steps:

(c) $f_0 \in \mathrm{Cn}_\mathfrak{b}(\mathcal{O}_\mathfrak{b}(h_\mathfrak{b} \cdot e))$ (by (a) and the def. of $\mathsf{OViol}_\mathfrak{b}$);

(d) $\exists C_0$ such that $C_0 \subseteq \mathcal{O}_\mathfrak{b}(h_\mathfrak{b} \cdot e)$ and $\mathsf{Causes}(C_0, f_0)$;

(e) $\exists f_1, C_1$ such that $f_1 \rightsquigarrow f_0$, $C_1 \rightsquigarrow C_0$ and $f_1 \in OCn_\mathfrak{b}(C_1)$ (by (d) and correctness of $OCn_\mathfrak{b}$);

(f) $C_0 \not\subseteq \mathcal{O}_\mathfrak{b}(h_\mathfrak{b})$ (otherwise $f_0 \in \mathsf{Violated}_\mathfrak{b}^\mathfrak{a}(h_\mathfrak{b})$, contradicting (b));

(g) $C_1 \in App\mathcal{O}_\mathfrak{b}(AppH_\mathfrak{b}(h' \cdot e)).New$ (by (d), (e), (f) and the correctness of $App\mathcal{O}_\mathfrak{b}$ and $AppH_\mathfrak{b}$);

(h) $f_1 \in AppSec(\mathfrak{b})$ ($f_1 \rightsquigarrow f_0$+ correctness of $AppSec(\mathfrak{b})$);

(i) $f_1 \in \mathsf{OViol}_\mathfrak{b}(AppH_\mathfrak{b}(h' \cdot e))$.

Claim 1 immediately follows.

Claim 2: $f_1 \notin \mathsf{UViol}_\mathfrak{b}(AppH_\mathfrak{b}(h'))$.

Suppose $f_1 \in \mathsf{UViol}_\mathfrak{b}(AppH_\mathfrak{b}(h'))$. We derive the following steps:

(j) $\exists C_2 \in App\mathcal{O}_\mathfrak{b}(AppH_\mathfrak{b}(h')).Nec$ such that $f_1 \in UCn_\mathfrak{b}(C_2)$ (by def. of $\mathsf{UViol}_\mathfrak{b}$);

(k) $\forall f_\mathfrak{b}$ such that $f_1 \rightsquigarrow f_\mathfrak{b}$, $f_\mathfrak{b} \in \mathrm{Cn}_\mathfrak{b}(\mathcal{O}_\mathfrak{b}(h_\mathfrak{b}))$ (by (j) and correctness of $UCn_\mathfrak{b}$ and $App\mathcal{O}_\mathfrak{b}$);

(l) $f_0 \in \mathrm{Cn}_\mathfrak{b}(\mathcal{O}_\mathfrak{b}(h_\mathfrak{b}))$ (from (k), since $f_1 \rightsquigarrow f_0$);

(m) $f_0 \in \mathsf{Violated}_\mathfrak{b}^\mathfrak{a}(h_\mathfrak{b})$ (from (l), since f_0 is a secret).

But (m) contradicts (b), so Claim 2 holds. From the above claims it follows immediately that $h_\mathfrak{a}$ is not approximately data secure. This completes the proof. ∎

6.8 Compact Approximations

In many applications (especially those where security checks are performed at runtime), the overhead caused by maintaining two approximate states for each client agent and computing two approximations of its consequence operation is unacceptable. Hence, we introduce a *compact* version of the approximate security check, where only the state after the answer and the overestimate of b's consequences need to be computed.

This has two advantages: first, the space needed to store the underestimate of b's consequences is saved, and second, the time needed to compute the underestimate of b's consequences as well as the time required to check if the secrets in the overestimate of b's consequences after the answer is a subset of the underestimate before the answer is saved. However, there is a price to pay, namely a decrease in the cooperativeness of the answer provided by agent a.

Definition 6.20 (Compact Approximation) *An approximation* $\mathbf{App}(\mathsf{b}) = \langle AppH_{\mathsf{b}}, App\mathcal{O}_{\mathsf{b}}, AppSec(\mathsf{b}), OCn_{\mathsf{b}}, UCn_{\mathsf{b}}\rangle$ *based on the languages* $pos\mathcal{H}_{\mathsf{b}}^{\mathsf{a}}$ *and* $App\mathcal{L}_{\mathsf{b}}$ *is* compact *if the following two conditions hold:*

1. *for all approximate histories* $h \in pos\mathcal{H}_{\mathsf{b}}^{\mathsf{a}}$, $App\mathcal{O}_{\mathsf{b}}(h).Nec = \emptyset$;
2. *for all* $C \subseteq App\mathcal{L}_{\mathsf{b}}$, $UCn_{\mathsf{b}}(C) = \emptyset$.

The following example shows a compact approximation of an agent b.

Example 6.11 *We return to Example 6.5. Suppose* tank1 *believes that it is possible that the* monitor *agent didn't know anything that can be expressed by* $App\mathcal{L}_{\mathsf{monitor}}^{\mathsf{tank1}}$ *when it was deployed and that the* monitor *agent does not believe anything it is told. Furthermore, it cannot infer anything from facts in* $App\mathcal{L}_{\mathsf{monitor}}^{\mathsf{tank1}}$. *In such a case, to underestimate the states and the consequence operations of the* monitor *agent, it uses the following: (1) for all* $h \in pos\mathcal{H}_{\mathsf{monitor}}^{\mathsf{tank1}}$, $App\mathcal{O}_{\mathsf{monitor}}(h).Nec = \emptyset$ *; (2) for all* $C \subseteq App\mathcal{L}_{\mathsf{monitor}}^{\mathsf{tank1}}$, $UCn_{\mathsf{monitor}}^{\mathsf{tank1}}(C) = \emptyset$.

Note that tank1*'s belief may be wrong and* tank1 *may know that there is a possibility that* monitor *knows more. As this possibility exists, for* tank1*'s approximation to be correct, it must be as described above.*

Note that in compact approximations, the underestimate of violated secrets prior to providing an answer is taken to be the empty set, and hence, the inclusion of Def. 6.19 is equivalent to:

$$\mathsf{OViol}_{\mathsf{b}}(AppH_{\mathsf{b}}(h' \cdot e)) = \emptyset \,.$$

As expected, this security condition depends only on one approximation of b's inferences, and only on the approximation of b's state *after* a's answer e.

The above equation immediately implies that compact approximations strengthen the notion of data security by requiring that no secret be derivable using a's answer. At first glance, this approach may appear similar to the naive security definition that requires b to derive no secret, no matter where it comes from (see Section 5.2). However, the paradoxical situation in which a's behavior is labeled non-secure because some other agent c has disclosed a secret is avoided by compact approximations. In fact, as $\mathsf{OViol}_\mathsf{b}$ only approximates only the inferences that are *caused* by a's answer, the secrets revealed by another agent, e.g. c, would not be included in $\mathsf{OViol}_\mathsf{b}$. The definition of correct overestimate (based on Causes) and the use of the field *New* in the definition of $\mathsf{OViol}_\mathsf{b}$ play a fundamental role in preserving this important property.

A nice property of compact approximations is that *every* correct approximation can be turned into a compact approximation which is correct! This is done via the following "compaction" operation.

Definition 6.21 (Compact Version) *The compact version of* $\mathbf{App}(\mathsf{b}) = \langle AppH_\mathsf{b},\ App\mathcal{O}_\mathsf{b},\ AppSec(\mathsf{b}),\ OCn_\mathsf{b},\ UCn_\mathsf{b}\rangle$ *is the compact approximation*

$$\mathbf{Compact}(\mathbf{App}(\mathsf{b})) = \langle AppH_\mathsf{b},\ \widehat{App\mathcal{O}_\mathsf{b}},\ AppSec(\mathsf{b}),\ OCn_\mathsf{b},\ (\lambda X.\emptyset)\rangle$$

where $\lambda X.\emptyset$ *is the constant function that always returns* $\emptyset$*, and for all* $h \in pos\mathcal{H}^\mathsf{a}_\mathsf{b}$,

$$\widehat{App\mathcal{O}_\mathsf{b}}(h) =_{def} \langle \emptyset,\ App\mathcal{O}_\mathsf{b}(h).Poss,\ App\mathcal{O}_\mathsf{b}(h).New\rangle\,.$$

The following result verifies that the compaction operator **Compact** preserves correctness.

Theorem 6.2 (Correctness Preservation) *If* $\mathbf{App}(\mathsf{b})$ *is correct, then* $\mathbf{Compact}(\mathbf{App}(\mathsf{b}))$ *is correct.*

Proof: By definition, $\mathbf{Compact}(\mathbf{App}_\mathsf{a}(\mathsf{b}))$ is correct if each of its components are correct. The correctness of $AppH_\mathsf{b}$, $AppSec(\mathsf{b})$ and OCn_b follows directly from the assumption that $\mathbf{App}_\mathsf{a}(\mathsf{b})$ is correct, since these components are shared by $\mathbf{Compact}(\mathbf{App}_\mathsf{a}(\mathsf{b}))$ and $\mathbf{App}_\mathsf{a}(\mathsf{b})$. The function $\lambda X.\emptyset$ satisfies trivially the correctness condition for underestimated consequence operations. Finally, the correctness of $\widehat{App\mathcal{O}_\mathsf{b}}$ depends on conditions 1-3 of Definition 6.9. Clearly, condition 1 is satisfied because $\widehat{App\mathcal{O}_\mathsf{b}}(h).Nec = \emptyset$ (by definition of **Compact**). Conditions 2 and 3 are satisfied because $\mathbf{App}_\mathsf{a}(\mathsf{b})$ is correct. This completes the proof. ∎

Replacing $\mathbf{App}(\mathsf{b})$ by $\mathbf{Compact}(\mathbf{App}(\mathsf{b}))$ may significantly improve performance. The price to be paid for this is a potential loss of cooperation. The following theorem says that whenever an agent a is approximately data secure w.r.t. a compact approximation of an agent b, then it is also approximately data secure w.r.t. the (perhaps uncompact) approximation of b.

Theorem 6.3 (Compact Approx. Security Implies Approx. Security) *If* h_a *is approximately data secure w.r.t.* $\mathbf{Compact}(\mathbf{App}(\mathsf{b}))$*, then* h_a *is approximately data secure w.r.t.* $\mathbf{App}(\mathsf{b})$.

Proof: Suppose h_a is approximately data secure w.r.t. $\mathbf{Compact}(\mathbf{App}_\mathsf{a}(\mathsf{b}))$ and let $h' \cdot e$ be an arbitrary prefix of h_a such that $e = \langle \mathsf{a}, \mathsf{b}, Ans\rangle$. Then, from the definition of data secure histories and compact histories it follows that:

$$\mathsf{OViol}_\mathsf{b}(AppH_\mathsf{b}(h' \cdot e)) = \emptyset,$$

where $\mathsf{OViol}_{\mathtt{b}}$ is defined w.r.t. **Compact**($\mathbf{App}_{\mathtt{a}}(\mathtt{b})$). Note also that $\mathbf{App}_{\mathtt{a}}(\mathtt{b})$ yields the same overestimation $\mathsf{OViol}_{\mathtt{b}}$ as **Compact**($\mathbf{App}_{\mathtt{a}}(\mathtt{b})$), because the components on which $\mathsf{OViol}_{\mathtt{b}}$ is based are the same in the two approximations. It follows that also under $\mathbf{App}_{\mathtt{a}}(\mathtt{b})$

$$\mathsf{UViol}_{\mathtt{b}}(AppH_{\mathtt{b}}(h')) \supseteq \mathsf{OViol}_{\mathtt{b}}(AppH_{\mathtt{b}}(h' \cdot e)) = \emptyset.$$

The above inclusion holds for arbitrary prefixes of $h_{\mathtt{a}}$; this implies that $h_{\mathtt{a}}$ is approximately data secure w.r.t. $\mathbf{App}_{\mathtt{a}}(\mathtt{b})$. ∎

As a consequence of this theorem, we know that to check whether a history $h_{\mathtt{a}}$ is approximately data secure w.r.t. $\mathbf{App}(\mathtt{b})$, it is sufficient to check whether $h_{\mathtt{a}}$ is approximately data secure w.r.t. **Compact**($\mathbf{App}(\mathtt{b})$).

Corollary 6.4 *For each history $h_{\mathtt{a}}$ which is approximately data secure w.r.t.* **Compact**($\mathbf{App}(\mathtt{b})$), *there exists a history $h'_{\mathtt{a}}$ which is approximately data secure w.r.t.* $\mathbf{App}(\mathtt{b})$ *and* $h_{\mathtt{a}} \leq^{\mathsf{coop}}_{\mathtt{a}} h'_{\mathtt{a}}$.

The converse of Theorem 6.3 (and Corollary 6.4) does not hold, in general, and therefore choosing to use **Compact**($\mathbf{App}(\mathtt{b})$) in place of $\mathbf{App}(\mathtt{b})$ may lead to a decrease in cooperation. This is demonstrated via the following example.

Example 6.12 *Consider the scenarios and approximations specified in examples 6.10, 6.9, 6.7 and 6.5. As discussed in Example 6.10 in the scenario in which the* monitor *agent received the soldier list from* com_c, *and e' is the event in which* tank1 *gives the* monitor *agent its soldier list, $h_{\mathtt{tank1}} \cdot e'$ is approximately data secure as*

$$\mathsf{UViol}_{\mathtt{monitor}}(h') = \mathsf{OViol}_{\mathtt{monitor}}(h' \cdot e') = \{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : soldiers())\}\,.$$

However, suppose we consider the compact version of the approximation described in Example 6.10. That is, the approximation described in Example 6.11 where: (1) for all $h \in pos\mathcal{H}^{\mathtt{tank1}}_{\mathtt{monitor}}$, $App\mathcal{O}_{\mathtt{monitor}}(h).Nec = \emptyset$; (2) for all $C \subseteq App\mathcal{L}^{\mathtt{tank1}}_{\mathtt{monitor}}$, $UCn^{\mathtt{tank1}}_{\mathtt{monitor}}(C) = \emptyset$.

Using this compact approximation,

$$\mathsf{UViol}_{\mathtt{monitor}}(h') = \emptyset$$

and

$$\mathsf{OViol}_{\mathtt{monitor}}(h' \cdot e') = \{\mathbf{in}(\mathtt{S}, \mathtt{tank1} : soldiers())\}\,.$$

Thus, $h' \cdot e'$ is not approximately data secure using the compact approximation. To make $h' \cdot e'$ approximately secure in this case, tank1 *should be less cooperative and not give the* monitor *agent its soldier list.*

6.9 Static Approximations

A *static* security check is one that checks upfront that an agent $\mathtt{a}$ is secure irrespective of what sequences of events may ensue (as long as those events are in accordance with the behavior of agent $\mathtt{a}$'s specification via its agent program, etc.). Unfortunately, the set of possible histories for $\mathtt{a}$– in general – is undecidable, as $\mathtt{a}$ can be as powerful as an arbitrary Turing machine (this is proved in the next section). Thus, static security checks can only be based on *approximate estimates* of $\mathtt{a}$'s possible future behaviors.

For this reason, the designer of agent $\mathfrak{a}$ must *overestimate* the set of possible histories that agent $\mathfrak{a}$ may indulge in so as to cover *at least* all the possible interactions between $\mathfrak{a}$ and an arbitrary agent $\mathfrak{b}$. If each such history in the overestimated set of possible histories is guaranteed to be secure at the time the agent is deployed, then security of $\mathfrak{a}$ is guaranteed upfront. The following definition says that a static agent approximation is one that takes into account such an overestimate of agent $\mathfrak{a}$'s possible space of histories.

Definition 6.22 (Static Agent Approximation, restriction, correctness) *A static approximation*
StaticApp($\mathfrak{b}$) *is an approximation of* $\mathfrak{b}$ *used by* $\mathfrak{a}$ *such that the domain of* $AppH_{\mathfrak{b}}$ *is extended to a set,* $pos\mathcal{H}_{\mathfrak{a}}^{+}$*, of histories for* $\mathfrak{a}$ *such that* $pos\mathcal{H}_{\mathfrak{a}}^{+} \supseteq pos\mathcal{H}_{\mathfrak{a}}$*. The set* $pos\mathcal{H}_{\mathfrak{a}}^{+}$ *will be referred to as the* approximation of $\mathfrak{a}$'s possible histories.

The dynamic restriction *of* **StaticApp**($\mathfrak{b}$) *is the agent approximation* **App**($\mathfrak{b}$) *obtained from* **StaticApp**($\mathfrak{b}$) *by restricting the domain of* $AppH_{\mathfrak{b}}$ *to* $pos\mathcal{H}_{\mathfrak{a}}$.

We say that **StaticApp**($\mathfrak{b}$) *is* correct *if all its components are correct. The correctness of* $AppH_{\mathfrak{b}}$ *is obtained by extending the correctness condition of Def. 6.3 to all* $h \in pos\mathcal{H}_{\mathfrak{a}}^{+}$*. The definition of correctness for the other components is unchanged.*

In the above definition, pos$\mathcal{H}_{\mathfrak{a}}^{+}$ is the "expanded" set of histories being considered in order to ensure (upfront) that agent $\mathfrak{a}$ is secure. The formal definition of static data security is given below.

Definition 6.23 (Static Data Security) *We say that the approximation* $pos\mathcal{H}_{\mathfrak{a}}^{+}$ *of* $\mathfrak{a}$*'s possible behaviors is* statically data secure w.r.t. **StaticApp**($\mathfrak{b}$) *if for all* $h \in pos\mathcal{H}_{\mathfrak{a}}^{+}$*,* h *is approximately data secure w.r.t.* **StaticApp**($\mathfrak{b}$).

Informally speaking, the following theorem guarantees that any agent known to be statically data secure is also data secure.

Theorem 6.5 (Static Security Preservation) *Let* **StaticApp**($\mathfrak{b}$) *be a correct static approximation of* $\mathfrak{b}$ *used by* $\mathfrak{a}$*. If* $pos\mathcal{H}_{\mathfrak{a}}^{+}$ *is statically data secure w.r.t.* **StaticApp**($\mathfrak{b}$)*, then* $\mathfrak{a}$ *is data secure w.r.t.* $\mathfrak{b}$.

Proof: First note that since **StaticApp**($\mathfrak{b}$) is correct, then its dynamic restriction $\mathbf{App}_{\mathfrak{a}}(\mathfrak{b})$ is also correct (straightforward from the definition). Now we prove the contrapositive of the theorem, which is equivalent. Suppose $\mathfrak{a}$ is *not* data secure w.r.t. $\mathfrak{b}$. Then, by Theorem 6.1, it is not approximately data secure w.r.t. $\mathbf{App}_{\mathfrak{a}}(\mathfrak{b})$. Consequently, some history $h_{\mathfrak{a}} \in \text{pos}\mathcal{H}_{\mathfrak{a}}$ is not data secure w.r.t. $\mathbf{App}_{\mathfrak{a}}(\mathfrak{b})$, and hence, for some of its prefixes $h' \cdot e$ such that $e = \langle \mathfrak{a}, \mathfrak{b}, Ans \rangle$,

$$\mathsf{UViol}_{\mathfrak{b}}(AppH_{\mathfrak{b}}(h')) \not\supseteq \mathsf{OViol}_{\mathfrak{b}}(AppH_{\mathfrak{b}}(h' \cdot e)). \qquad (*)$$

By definition of static approximation, pos$\mathcal{H}_{\mathfrak{a}}^{+} \supseteq$ pos$\mathcal{H}_{\mathfrak{a}}$, so $h_{\mathfrak{a}} \in$ pos$\mathcal{H}_{\mathfrak{a}}^{+}$. It follows (by (*)) that pos$\mathcal{H}_{\mathfrak{a}}^{+}$ is not statically data secure w.r.t. **StaticApp**($\mathfrak{b}$). ∎

The following theorem says that static security implies data security w.r.t. $\mathfrak{a}$'s dynamic restriction. It also proves that static checks are stricter, i.e., some agents are approximately data secure but not statically data secure.

Theorem 6.6 (Static vs. Dynamic Verification)

1. *Under the hypotheses of Theorem 6.5, if* $pos\mathcal{H}_{\mathfrak{a}}^{+}$ *is statically data secure, then* $\mathfrak{a}$ *is approximately data secure w.r.t.* **StaticApp**$(\mathfrak{b})$*'s dynamic restriction.*

2. *There exists an agent* $\mathfrak{a}$ *and a correct static approximation* **StaticApp**$(\mathfrak{b})$ *based on* $\mathfrak{a}$*'s history approximation* $pos\mathcal{H}_{\mathfrak{a}}^{+}$*, such that* $\mathfrak{a}$ *is approximately data secure w.r.t.* **StaticApp**$(\mathfrak{b})$*'s dynamic restriction, but* $pos\mathcal{H}_{\mathfrak{a}}^{+}$ *is not statically data secure w.r.t.* **StaticApp**$(\mathfrak{b})$*.*

Proof: The proof of part 1 is contained in the proof of Theorem 6.5 (there we proved that if some $h_{\mathfrak{a}}$ is not approximately data secure w.r.t. the dynamic approximation $\mathbf{App}_{\mathfrak{a}}(\mathfrak{b})$, then $\mathrm{pos}\mathcal{H}_{\mathfrak{a}}^{+}$ is not statically data secure w.r.t. **StaticApp**$(\mathfrak{b})$).

To prove part 2, suppose $\mathfrak{b}$ is the agent defined in the proof of Proposition 5.2, and let $\mathbf{App}_{\mathfrak{a}}(\mathfrak{b})$ be any correct approximation of $\mathfrak{b}$ with $AppSec(\mathfrak{b}) \neq \emptyset$ (we can choose $AppSec(\mathfrak{b})$ arbitrarily). Let $\mathrm{pos}\mathcal{H}_{\mathfrak{a}}$ be the set of all histories h_n illustrated in the proof of Proposition 5.2, with the further requirement that $Ans_i = \emptyset$ for all $i > 0$, so that $\mathfrak{a}$ is trivially data secure. Now let f be any secret in $Sec_{\mathfrak{a}}(\mathfrak{b})$. Define $\mathrm{pos}\mathcal{H}_{\mathfrak{a}}^{+} = \mathrm{pos}\mathcal{H}_{\mathfrak{a}} \cup \{\langle\langle \mathfrak{a}, \mathfrak{b}, \{f\}\rangle\rangle\}$. Note that $\mathbf{App}_{\mathfrak{a}}(\mathfrak{b})$ is the dynamic restriction of **StaticApp**$(\mathfrak{b})$. Clearly, $\mathrm{pos}\mathcal{H}_{\mathfrak{a}}^{+}$ is not statically data secure w.r.t. **StaticApp**$(\mathfrak{b})$ ($\mathfrak{b}$ believes the secret f and stores it in its state). This completes the proof. ▮

7 Undecidability Results

As stated above, the developer of an agent may be interested in two types of security verification methods.

Static security verification: In this mode of security verification, the agent developer would like to be sure, when deploying an agent, that the agent will always be secure. Such security verification can be performed once and for all at the time the agent is deployed, and leads to no run-time security verification. Thus, once an agent is known to be statically secure, no run-time security checks are needed.

Dynamic security verification: In this mode of security verification, no security checks are made at the time the agent is deployed. Rather, every time the agent receives a request, a run-time security check is made.

As mentioned in the preceding section, we will show that it is impossible to decide statically whether an agent is approximately data secure. The first result below states that even the relatively simple notion of surface security is undecidable.

Theorem 7.1 (Undecidability of Surface Security) *The problem of deciding statically whether an arbitrary IMPACT agent is surface secure is undecidable.*

Proof: We prove this theorem by uniformly reducing the halting problem for arbitrary deterministic Turing machines M to a surface security verification problem. For this purpose, we simulate M with a suitable agent $\mathfrak{a}$ that outputs a secret f when a final state of M is reached.

Recall that M's *configuration* consists of the tape contents plus the current state of M's *finite control*, which in turn is a set of 5-tuples of the form

$$\langle s, v, v', s', m\rangle$$

where s is the current state, v is the symbol under M's head, v' is the symbol to be overwritten on v, s' is the next state and $m \in \{\texttt{left}, \texttt{right}\}$ specifies the head's movement. We assume M's configuration is encoded by means of a suitable package **TMC** (which stands for Turing Machine Configuration), which provides code calls for updating M's configuration and two code calls $\mathbf{TMC}: \mathit{current\_symbol}()$ and $\mathbf{TMC}: \mathit{current\_state}()$ to read the symbol pointed to by the machine's head and the machine's current state, respectively.[3]

We also assume the agent has an action $\mathit{move}(v', s', m)$ (implemented with **TMC**'s code calls for updating M's configuration), that simulates one move, i.e. it sets the current tape symbol to v', it sets the current state to s' and moves the head as specified by m. The finite control of M will be modeled through a suitable agent program that specifies under what conditions the *move* action has to be executed.

For each 5-tuple $\langle s, v, v', s', m\rangle$ in the finite control there is a corresponding agent program rule **R** like the following:

$$\begin{aligned}&\mathbf{O}\,\mathit{move}(v', s', m) \leftarrow\\ &\quad \mathbf{in}(s, \mathbf{TMC}: \mathit{current\_state}())\ \ \&\\ &\quad \mathbf{in}(v, \mathbf{TMC}: \mathit{current\_symbol}())\,.\end{aligned}$$

Intuitively, this rule causes replacement of the current configuration of M with the new one specified by v', s' and m.

Finally, for each final state s of M, $\mathtt{a}$'s agent program contains a rule

$$\mathbf{O}\,\mathit{send}(\mathtt{b}, f) \leftarrow \mathbf{in}(\mathtt{s}, \mathbf{TMC}: \mathit{current\_state}())\,.$$

where f is a secret and $\mathit{send}(b, f)$ is an action that sends the answer $\{f\}$ to $\mathtt{b}$.

Clearly, by construction, $\mathtt{a}$ outputs a secret f (thereby violating surface security) if and only if M terminates. This completes the proof. ∎

Remark 7.1 *All that is needed to simulate a Turing machine is a package with a dynamic data structure (i.e. a data structure whose size is not known at compile time). In [53], we encode the Turing machine configuration with a standard IMPACT package originally designed to encode meta-knowledge about other agents. Turing machine configurations could also be encoded using a DBMS package.*

An immediate consequence of the above result is that checking data security is undecidable.

[3]Note that such a package can be easily implemented in any modern programming language, by maintaining two variables that encode the current tape symbol and the current state, and two linked lists of symbols that encode the used portions of the tape on the left and on the right of M's head, respectively. Clearly, IMPACT agents are expected to use packages of this sort, as well as much more complicated packages.

Corollary 7.2 *The problems of deciding statically whether an arbitrary IMPACT agent is data secure or approximately data secure, are undecidable.*

Proof: Immediate from theorems 5.2 and 7.1. ∎

The previous undecidability results also may be easily extended to show that action security is undecidable.

Theorem 7.3 (Undecidability of Action Security) *The problem of deciding statically whether an arbitrary IMPACT agent is action secure is not decidable.*

Proof: Similar to the proof of Theorem 7.1. An arbitrary Turing machine M can be encoded into an *IMPACT* agent as shown in the proof of Theorem 7.1. However, the rules that output a secret when a final state of M is reached are replaced by rules that do a forbidden action. Then the halting problem is reduced to action security verification. ∎

The above results show that given an arbitrary agent and its security needs as input, statically ensuring that the agent is secure is undecidable. As we will show later in Section 8, all is not lost. Two important facts are not ruled out by the above (seemingly depressing) undecidability results.

1. First, it will be possible to find *sufficient* conditions that can be checked statically on an agent and its security needs. If these conditions are satisfied, then the agent is approximately data secure. Note that the converse is not true — there may be agents that are approximately data secure and do not satisfy these (sufficient) conditions.
2. Second, it will turn out that dynamic security verification is in fact decidable, though the run-time cost of checking dynamic security can adversely affect system performance. Actually, the main reason for the undecidability results is that it is impossible to predict $\mathfrak{a}$'s behavior; run-time checks, on the contrary, need no prediction – they only have to inspect the outgoing messages, as they are generated by $\mathfrak{a}$.

In the rest of this paper, we will describe mechanisms through which the designer of an agent may articulate how his agent approximates other agents, and then we will show how these articulations may be checked for static/dynamic security.

8 Security Specification Languages

In this section, we will provide a "tight" language within which the developer of an agent $\mathfrak{a}$ can express the approximations that $\mathfrak{a}$ must use. This language consists of two components:

History component $\mathbf{Hist}_{\mathfrak{a}}$. This component is used to record and maintain $\mathfrak{a}$'s history, $h_{\mathfrak{a}}$.

Agent approximation program $\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}$. This is a set of rules that encode $\mathfrak{a}$'s approximation of $\mathfrak{b}$, denoted by $\mathbf{App}(\mathfrak{b})$ in the abstract approximation framework (cf. Definition 6.16).

Once these languages are defined, in Section 9, we will define a package called SecP_a that may be used to maintain, compile and execute the programs that perform static, dynamic and combined security checks. We now discuss each of these components below.

8.1 The History Component $\mathbf{Hist}_a$

The developer of an agent a needs to answer the following questions pertaining to the history maintained by her agent:

- *Which events should be stored in the historical archive?* In general, an agent a may choose to store only certain types of events. This may increase the efficiency of history manipulation, but may decrease the quality of the *approximations* of other agents' histories, which are based on a's own history (see the examples in Section 6.2).
- *Which attributes of these events should be stored?* Possible attributes of an answer message which an agent developer might wish to store are the requested service, the receiver, the answer, the time at which the answer was sent.

The history component may be viewed as a software package (cf. Section 3) which stores a totally ordered list of time-stamped events, that can be queried and updated by means of the following functions.

- `retrieve_reqs(Sender,Receiver,Request,When)`: Retrieve all stored request messages sent by `Sender` to `Receiver` at time `When`, and which match `Request`. Parameter `When` has the form `Op<time>`, where `Op` is one of $<, \leq, =, \neq, \geq, >$. The above parameters may be left unspecified, in part or entirely, using the wildcard '_'. For example, the invocation `retrieve_reqs(b,_,_,> 20:jan:95:7pm)` retrieves all stored request messages sent by b after the specified time.
- `retrieve_answ(Sender,Receiver,Fact,When)`: Retrieve all stored answer messages sent by `Sender` to `Receiver` at time `When`, and such that the answer contains a fact f which matches `Fact`. The variables of `Fact` are instantiated with f. `When` can be specified as explained above; wildcards may be used.
- `retrieve_actn(Act,When)`: Retrieve all stored actions that match `Act`, and executed at the time specified by `When`. The action name and/or its arguments may be left unspecified using the wildcard '_'.

The history package is completed by the *history update actions* described below.

- `insert_reqs(Sender,Receiver,Req,When)`, `insert_answ(Sender,Receiver,Ans,When)`, `insert_actn(Act,When)`: These actions append a new event to a's history.
- `delete(Event)`: Deletes `Event` from the history .

Note that the history component of an IMPACT agent may be viewed as just another data structure together with the above set of associated functions. Hence, the concepts of code call and code call conditions apply directly to the history component .

Definition 8.1 (History Conditions) *Suppose RF is one of the above three retrieval functions, and* **args** *is a list of arguments for RF of the appropriate type. We may inductively define history conditions as follows.*

- $\mathbf{in}(\texttt{X}, \mathit{Hist}_{\mathfrak{a}} : RF(\texttt{args}))$ *is a history condition.*
- *If Op is any of $<, \leq, =, \neq, \geq, >$, and T_1, T_2 are variables or objects, then T_1 Op T_2 is a history condition.*
- *If χ_1, χ_2 are history conditions then $(\chi_1 \,\&\, \chi_2)$ is a history condition.*

The syntactic restrictions obeyed by history conditions will be needed in Section 8.2. In general, $\mathsf{Hist}_{\mathfrak{a}}$'s functions may occur side by side with arbitrary conditions. The following example presents some history conditions that an agent in the Tank Example might use.

Example 8.1 *The following are history conditions which can be used by* tank1.

$$\begin{array}{r} \mathbf{in}(\texttt{Event1}, \mathit{Hist}_{\mathfrak{a}} : \mathit{retrieve\_reqs}(\texttt{monitor}, \texttt{tank1}, \_, > 20{:}june{:}1995)())\ \& \\ \mathbf{in}(\texttt{Event2}, \mathit{Hist}_{\mathfrak{a}} : \mathit{retrieve\_reqs}(\texttt{com\_c}, \texttt{tank1}, \_, > 20{:}june{:}1995)())\& \\ \mathit{Event1.req} = \mathit{Event2.req} \end{array}$$

The above history condition retrieves all the requests that were sent both by the monitor *agents and the* com_c *agent after June, 20th 1995.*

8.2 Agent Approximation Languages

We are now ready to explain how the designer of agent $\mathfrak{a}$ approximates other agents. To do so, the designer of $\mathfrak{a}$ writes one set of rules for each component of $\mathfrak{a}$'s approximation of $\mathfrak{b}$. Specifically,

1. The designer first writes a set of rules called *history approximation rules* through which the designer specifies how designed agent approximates the history of another agent;
2. Then, the designer writes a set of *state approximation rules* which specifies how the designed agent approximates the state of another agent;
3. Then, the designer writes a set of *consequence approximation rules* that specify how the agent captures the approximate consequence operation of another agent;
4. Finally, the designer writes a set of *secrets approximation rules* specifying the set of approximate secrets.

8.2.1 History Approximation

We now discuss how history approximations may be expressed by an agent developer in *IMPACT*. This is done through a construct called a history constraint that is defined via two simpler constructs defined below.

Definition 8.2 (Pure History Constraint) `requested(Sender,Receiver,Request,Time)`, `told(Sender,Receiver,Answer,Time)`, *and* `done(Agent,ActionName,Time)` *are called* pure history constraints.

Pure history constraints correspond to the three possible event types — request messages, answer messages and action events. The argument `Time` is a number which denotes the time at which the event happened. The other kind of history constraint is a comparison constraint.

Definition 8.3 (Comparison Constraint) *If T_1, T_2 are either objects or variables, and Op is one of the comparison operators $<, \leq, =, \neq, \geq, >$, then T_1 Op T_2 is called a* comparison constraint.

We are now ready to define history constraints.

Definition 8.4 (History Constraint) *A* history constraint *is either a comparison constraint or a pure history constraint.*

The reader is cautioned that history constraints and history conditions (defined earlier) are two different concepts ! We are now ready to provide examples of history constraints associated with the Tank Example.

Example 8.2 *The following expressions are pure history constraints:*

$$\begin{aligned}\texttt{requested}(\mathsf{com\_c}, \mathsf{tank1}, \textit{\textbf{set:speed}}(\textit{new\_speed}), 20 : \mathsf{june} : 1999),\\ \texttt{requested}(\mathsf{monitor}, \mathsf{tank1}, \textit{\textbf{fuel\_level}}(), \mathtt{X}_{now} - 60),\\ \texttt{told}(\mathsf{tank1}, \mathsf{monitor}, \mathbf{in}(\mathtt{low}, \mathsf{tank1} : \textit{\textbf{fuel\_level}}(\mathtt{X}_{now})), \mathtt{X}_{now}),\\ \texttt{done}(\mathsf{tank1}, \textit{set\_speed}(\textit{55kmh}), 15 : 00 : 20 : \mathsf{june} : 1999)\,.\end{aligned}$$

The following expressions are comparison constraints: $T_1 < \mathtt{X}_{now} - 5$, $T_1 = T_2$, and $T_3 \neq 15 : 00 : 20 : june : 1999$.

Definition 8.5 (History Approximation Program) *A* history approximation program *(used by agent $\mathfrak{a}$ for agent $\mathfrak{b}$) is a set R_{his}, of rules of the form*

$$PHC \leftarrow \chi_{\mathsf{hist}},$$

where PHC is a pure history constraint and χ_{hist} is a history condition (not a history constraint!).

When the developer of agent $\mathfrak{a}$ wishes to approximate the history of agent $\mathfrak{b}$, he or she explicitly specifies a history approximation program, R_{his}, which implicitly specifies a set of histories that "satisfy" the rules in R_{his}, and this set of histories reflects agent $\mathfrak{a}$'s approximation of the histories of agent $\mathfrak{b}$.

Definition 8.6 (History Satisfaction) *Let $h_{\mathfrak{b}} = \langle e_1, e_2, \ldots, e_i, \ldots \rangle$ be a history for $\mathfrak{b}$. $h_{\mathfrak{b}}$* satisfies *a conjunction of history constraints, HC, if there is a ground instance $HC\theta$ of HC such that:*

- *each comparison constraint in $HC\theta$ is true;*
- *each pure history constraint $c \in HC\theta$* matches *some event e in $h_{\mathtt{b}}$ of the corresponding type, in the sense that the fields* ***Sender, Receiver, Request*** *and* ***Answer*** *of c equal the corresponding elements of e;*
- *the parameters* `Time` *correctly reflect the ordering of the events; formally, for all pure history constraints c' and c'' in $HC\theta$, whose last parameters are $\mathtt{Time}'$ and $\mathtt{Time}''$, respectively, and such that c' and c'' match events e_j and e_k of $h_{\mathtt{b}}$, (respectively), it holds that $\mathtt{Time}' \leq \mathtt{Time}'' \Leftrightarrow j \leq k$.*

The following example shows some histories in the Tank example and some history constraints that are satisfied.

Example 8.3 *A history $h_{\mathtt{monitor}}$ for* monitor *of the form*

$\langle \ldots \langle \mathtt{monitor}, \mathtt{tank1}, \mathit{location}(\mathtt{X}_{now}) \rangle \ldots \langle \mathtt{tank1}, \mathtt{monitor}, \mathbf{in}((50,20,40), \mathtt{tank1} : \mathit{location}(\mathtt{X}_{now}))\} \rangle \ldots \rangle$

(containing a service request and the corresponding answer) satisfies the history constraints:

$$\mathtt{requested}(\mathtt{monitor}, \mathtt{tank1}, \mathit{location}(\mathtt{X}_{now}), \mathtt{T1}),$$
$$\mathtt{told}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}((50,20,40), \mathtt{tank1} : \mathit{location}(\mathtt{T2}))),\ \mathtt{T1} \leq \mathtt{T2}.$$

Given a history approximation program, R_{his}, used by agent $\mathtt{a}$ to approximate the history of agent $\mathtt{b}$, the abstract approximation of agent $\mathtt{b}$'s history may now be stated intuitively as follows:

1. Find all pairs $(HC, \chi_{\mathsf{hist}})$ where HC is a conjunction of history constraints such that repeatedly unfolding (i.e. replacing the pure history constraints in HC by the bodies of rules whose heads unify with the pure history constraint) HC against the rules in R_{his} yields the history condition χ_{hist}.
2. For each such pair $(HC, \chi_{\mathsf{hist}})$, let θ be the composition of all the unifying substitutions involved in the previous step.
3. For each substitution σ such that $\chi_{\mathsf{hist}}\sigma$ is true in the current state of the history component, $HC\theta\sigma$ is possibly satisfied by a history of agent $\mathtt{b}$.
4. Any history that satisfies $HC\theta\sigma$ is considered to be a possible history of $\mathtt{b}$ by $\mathtt{a}$.

The following formal definitions formalize this point.

Definition 8.7 (Resolvent, Derivation) *Let G be a conjunction of atoms $A_1 \& \ldots \& A_n$, and let $r = (H \leftarrow B)$ be a rule whose head can be unified with some A_i $(1 \leq i \leq n)$, with a substitution θ. The* resolvent *of G and r w.r.t. θ (with* selected literal *A_i) is $(A_1 \& \ldots \& A_{i-1} \& B \& A_{i+1} \& \ldots \& A_n)\theta$.*

A standardized apart member *of a set of rules R is a variant of a rule $r \in R$, obtained by uniformly renaming R's variables with fresh variables, never used before.*

A derivation *from a set of rules R with substitutions $\theta_1 \dots \theta_m$ is a sequence $G_0, \dots, G_m$ such that for all $i = 1 \dots n$, G_i is a resolvent of G_{i-1} and some standardized apart member r_i of R, w.r.t. θ_i. If $G_0, \dots, G_m$ is a derivation from R with substitutions $\theta_1 \dots \theta_m$, and θ is the composition of $\theta_1 \dots \theta_m$, then we write*

$$G_0 \longmapsto_R^\theta G_m \,.$$

If the θ_is are all most general unifiers, then we write $G_0 \overset{\mathsf{mg}\ \theta}{\longmapsto}_R G_m$.

The reader is warned that $\longmapsto$ does *not* denote logical implication, but rather goal-rewriting. In fact, $G_0 \longmapsto_R^\theta G_m$ means that by repeatedly applying the rules of R to the initial goal G_0 in a top-down (or backward-chaining) fashion, G_m can be obtained at some point. The relation between $\longmapsto$ and logical implication is the following: if $G_0 \longmapsto_R^\theta G_m$ holds, then G_m and R *imply* $G_0\theta$ (in symbols: $G_0\theta \leftarrow G_m \wedge (\bigwedge R)$).

Thus, in particular, $G_0 \longmapsto_R^\theta G_m$ might hold, but $G_0 \,\&\, G_0' \longmapsto_R^{\theta'} G_m$ might not because the rules in R might not be sufficient to rewrite G_0' to G_m or a subset thereof. Using this concept, we may now precisely specify the approximate histories of $\mathfrak{b}$ used by agent $\mathfrak{a}$ as follows.

$$AppH_{\mathfrak{b}}(h_{\mathfrak{a}}) =_{def} \{HC\theta\sigma \mid HC \longmapsto_{R_{\mathsf{his}}}^\theta \chi_{\mathsf{hist}} \text{ and } \sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})\}. \qquad (2)$$

The following example uses the Tank Example to illustrate how an agent $\mathfrak{a}$ might approximate the history of agent $\mathfrak{b}$.

Example 8.4 *Let us consider the approximation of the history of the* monitor *agent by* tank1 *in the Tank Example.* tank1 *does not have a lot of information on the interactions of the* monitor *agent with other agents and its actions. Even the set of agents contacted by the* monitor *agent is not known. Some of them might know* tank1*'s region or its fuel level at different times and disclose it to the* monitor *agent. This may be expressed via the following history approximation rules. They say that for all* $\mathtt{X} \neq$ tank1 *and* $\mathtt{T} \leq \mathtt{T1}$, monitor*'s history may contain messages from* X *to* monitor*, specifying* tank1*'s region or fuel level, or both.*

$$\begin{aligned}
&(\mathbf{r1}) \quad \mathtt{told}(\mathtt{X}, \mathrm{monitor}, \mathbf{in}(\mathtt{R}, \mathrm{tank1} : \boldsymbol{\mathit{region}}(\mathtt{T})), \mathtt{T1}) \;\leftarrow\; \mathtt{X} \neq \mathrm{tank1} \;\&\; \mathtt{T} \leq \mathtt{T1}\,. \\
&(\mathbf{r2}) \quad \mathtt{told}(\mathtt{X}, \mathrm{monitor}, \mathbf{in}(\mathtt{L}, \mathrm{tank1} : \boldsymbol{\mathit{fuel\_level}}(\mathtt{T})), \mathtt{T2}) \;\leftarrow\; \mathtt{X} \neq \mathrm{tank1} \;\&\; \mathtt{T} \leq \mathtt{T2}\,.
\end{aligned}$$

The only assumption we make here is that the agents involved in this scenario do not talk about the future. Only old or current region information and fuel levels are communicated. This is expressed by $\mathtt{T} \leq \mathtt{T1}$ *and* $\mathtt{T} \leq \mathtt{T2}$.

We assume that in some cases, tank1 *itself may disclose its old fuel levels to the* monitor *agent. We also assume that* tank1 *keeps all its answers in* $\mathsf{Hist}_{\mathrm{tank1}}$ *for only one hour, then deletes them. Then, a recent answer can be in* monitor*'s history only if a corresponding message is stored in* $\mathsf{Hist}_{\mathrm{tank1}}$*, while older messages may be in the* monitor *agent's history* *regardless of* $\mathit{Hist}_{\mathtt{tank1}}$*'s contents. This can be expressed via the following rules.*

$$
\begin{aligned}
&(\mathbf{r3})\quad \mathtt{told}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T})), \mathtt{T3}) \leftarrow \\
&\qquad \mathbf{in}(\mathtt{Ev}, \mathit{Hist}_{\mathtt{tank1}} : \mathit{retrieve\_answ}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T})), \_)) \ \& \\
&\qquad \mathtt{T3} \geq \mathtt{Ev.time}\,. \\
&(\mathbf{r4})\quad \mathtt{told}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T})), \mathtt{T4}) \leftarrow \\
&\qquad \mathtt{T} \leq \mathtt{T4} \ \& \\
&\qquad \mathtt{T4} \leq \mathtt{now} - 60\,.
\end{aligned}
$$

Rule $(\mathbf{r3})$ *states that an answer message from* tank1 *may be in* monitor*'s history if there is a corresponding message* Ev *in* tank1*'s history (second line). Message delivery might not be instantaneous; there may be a delay before the answer is received by* monitor *(third line).*

Rule $(\mathbf{r4})$ *is needed because events older than 60 minutes are deleted from* $\mathit{Hist}_{\mathtt{tank1}}$.[4] *Therefore, if* $\mathtt{T4} \leq \mathtt{now} - 60$*, then an answer message from* tank1 *may be in* monitor*'s history while the corresponding event has been deleted from* $\mathit{Hist}_{\mathtt{tank1}}$. *Condition* $\mathtt{T} \leq \mathtt{T4}$ *says that L refers to a time point earlier than the answer delivery time. This condition is useless in* $(\mathbf{r3})$*, because* tank1 *cannot return a future fuel level, and hence,* $\mathtt{T} \leq \mathtt{Ev.time} \leq \mathtt{T3}$.

If R_{his} *consists of the rules* $(\mathbf{r1})$*-*$(\mathbf{r4})$ *above, and*

$$
\begin{aligned}
HC \;=\; &\mathtt{told}(\mathtt{X}, \mathtt{monitor}, \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T})), \mathtt{T}') \ \& \\
&\mathtt{told}(\mathtt{Y}, \mathtt{monitor}, \mathbf{in}(\mathtt{R}, \mathtt{tank1} : \mathit{region}(\mathtt{T})), \mathtt{T}'')
\end{aligned}
$$

then there exist three derivations $HC \longmapsto^{\theta_i}_{R_{\mathsf{his}}} \chi^i_{\mathsf{hist}}$ $(i = 1, 2, 3)$.

The first one applies $(\mathbf{r1})$ *and* $(\mathbf{r2})$*, and yields:*

$$
\begin{aligned}
\chi^1_{\mathsf{hist}} &= \mathtt{X} \neq \mathtt{tank1} \ \& \ \mathtt{T} \leq \mathtt{T}' \ \& \ \mathtt{Y} \neq \mathtt{tank1} \ \& \ \mathtt{T} \leq \mathtt{T}'', \\
HC\theta_1 &= HC\,.
\end{aligned}
$$

This means that (it is estimated that) the monitor *agent's history may contain two events matching* HC*, provided that* χ^1_{hist} *holds.*

The other derivations use $(\mathbf{r1})$ *and one of* $(\mathbf{r3})$ *and* $(\mathbf{r4})$*, yielding:*

$$
\begin{aligned}
\chi^2_{\mathsf{hist}} &= \mathbf{in}(\mathtt{Ev}_1, \mathit{Hist}_{\mathtt{tank1}} : \mathit{retrieve\_answ}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T})), \_)) \ \& \\
&\quad \mathtt{T}' \geq \mathtt{Ev}_1\mathtt{.time} \ \& \\
&\quad \mathtt{Y} \neq \mathtt{tank1} \ \& \ \mathtt{T} < \mathtt{T}'', \\
HC\theta_2 &= \mathtt{told}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}(\mathtt{L}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T})), \mathtt{T}') \ \& \\
&\quad \mathtt{told}(\mathtt{Y}, \mathtt{monitor}, \mathbf{in}(\mathtt{R}, \mathtt{tank1} : \mathit{region}(\mathtt{T})), \mathtt{T}'')\,;
\end{aligned}
$$

$$
\begin{aligned}
\chi^3_{\mathsf{hist}} &= \mathtt{T} \leq \mathtt{T}' \ \& \ \mathtt{T}' \leq \mathtt{now} - 60 \ \& \ \mathtt{Y} \neq \mathtt{tank1} \ \& \ \mathtt{T} < \mathtt{T}'', \\
HC\theta_3 &= HC\theta_2\,.
\end{aligned}
$$

[4]Note the agent's state is changing over time (Definition 3.2). That is, the values of its data objects may change over time. In particular, the value of now is changing over time.

Again, this means that the $\mathtt{monitor}$ *agent's history may contain two events that match* $HC\theta_i$ *if the corresponding condition* χ^i_{hist} *is satisfied. For* $i = 2$*, checking such condition involves inspecting* $\mathtt{tank1}$*'s history* $\mathit{Hist}_{\mathtt{tank1}}$*. This can be done either dynamically (at run time) or statically, by estimating how the history condition will be evaluated in the future as discussed below.*

8.2.2 State Approximation Language

We are now ready to define how an agent $\mathtt{a}$ approximates the state of another agent $\mathtt{b}$. Such an approximation has three fields, *Nec*, *Poss* and *New*, that capture (respectively) the conditions which are deemed to be necessarily true, possibly true, possibly true *and* caused by the last event in $\mathtt{b}$'s history. We will only consider compact approximations where *Nec* is empty. In order to express the set *Poss*, the agent developer writes a set of rules called the *state approximation program.*

Definition 8.8 (State approximation program) *The state approximation program used by* $\mathtt{a}$ *to approximate the state of agent* $\mathtt{b}$ *is a finite set of rules of the form*

$$\mathcal{B}_{\mathtt{a}}(\mathtt{b}, f) \leftarrow HC,$$

where f *is a fact from the approximate fact language* $App\mathcal{L}_{\mathtt{b}}$ *and* HC *is a set of history constraints.*

Intuitively, the above rule says that if $\mathtt{b}$'s history satisfies HC, then f *might* be in $\mathtt{b}$'s state.

By analogy with the implementation of history approximations, the relation between the abstract notion of state approximation and the corresponding program rules is given by the following two equations.

$$App\mathcal{O}_{\mathtt{b}}(H).Poss =_{def} \{B\theta\sigma \mid B \longmapsto^{\theta}_{R_{\mathsf{sta}}} HC \text{ and } HC\sigma \in H\}\,. \tag{3}$$

This is perfectly analogous to equation (2). The definition of the "*.New*" field is slightly more complex:

$$\begin{aligned} &App\mathcal{O}_{\mathtt{b}}(AppH_{\mathtt{b}}(h_{\mathtt{a}})).New =_{def} \\ &\quad \{B\theta\sigma \mid B \longmapsto^{\theta}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi_{\mathsf{hist}},\ \sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}}),\ \text{some} \\ &\quad \mathbf{in}(E, \mathsf{Hist}_{\mathtt{a}} : retrieve\_answ(\mathtt{a}, \mathtt{b}, \ldots)) \text{ belongs to } \chi_{\mathsf{hist}}\sigma \\ &\quad\ \text{and } E \text{ is the last event of } \mathsf{Hist}_{\mathtt{a}}\}\,. \end{aligned} \tag{4}$$

(Recall that the "*Nec*" field is not needed for compact approximations.)

The difference between the above two definitions can be explained as follows: possibly "*New*" facts are identified by extending the derivations down to code call conditions χ_{hist}, using R_{his}; if such code call conditions refer to the last event E stored in $\mathsf{Hist}_{\mathtt{a}}$, then the given fact B might have been caused by such E, and for this reason, B might be a new fact. Conversely, if B does never need event E to be derived, then clearly B cannot be caused by E (according to our approximate knowledge) and hence it cannot be new.

Given a state approximation program R_{sta}, the approximate state of agent $\mathtt{b}$ specified by agent $\mathtt{a}$ is given by the following proposition.

Proposition 8.1 *$AppO_{\mathsf{b}}(AppH_{\mathsf{b}}(h_{\mathsf{a}})).Poss = \{B\theta\sigma \mid B \longmapsto^{\theta}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi_{\mathsf{hist}}$ and $\sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})\}$, where χ_{hist} ranges over history code call conditions.*

Proof: First we prove the left-to-right inclusion. Assume that $B_0 \in AppO_{\mathsf{b}}(AppH_{\mathsf{b}}(h_{\mathsf{a}})).Poss$. By (3), this means that B_0 has the form $B\theta\sigma$ and for some history constraints HC, there is a derivation $B \longmapsto^{\theta}_{R_{\mathsf{sta}}} HC$ where $HC\sigma \in AppH_{\mathsf{b}}(h_{\mathsf{a}})$.

This membership, by (2), implies that $HC\sigma$ has the form $HC'\theta'\sigma'$ and there is a derivation $HC' \longmapsto^{\theta'}_{R_{\mathsf{his}}} \chi_{\mathsf{hist}}$ for some set of history constraints $\chi_{\mathsf{hist}}$with $\sigma' \in \mathsf{Sol}(\chi_{\mathsf{hist}})$.

By combining the ground instances of the two derivations we obtain a derivation $B_0 \longmapsto^{\theta_1}_{R_{\mathsf{sta}}} HC\sigma \longmapsto^{\theta_2}_{R_{\mathsf{his}}} \chi_{\mathsf{hist}}\sigma'$, and hence, by setting $\theta'' = \theta_1 \circ \theta_2$, $\chi''_{\mathsf{hist}} = \chi_{\mathsf{hist}}\sigma'$ and $\sigma'' = \epsilon$, where ϵ denotes the empty substitution, we obtain:

$$B_0 \longmapsto^{\theta''}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi''_{\mathsf{hist}},$$

where $\sigma'' \in \mathsf{Sol}(\chi''_{\mathsf{hist}})$. As a consequence,

$$B_0 \in \{B\theta\sigma \mid B \longmapsto^{\theta}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi_{\mathsf{hist}} \text{ and } \sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})\}.$$

Since B_0 is an arbitrary member of $AppO_{\mathsf{b}}(AppH_{\mathsf{b}}(h_{\mathsf{a}})).\,poss$, this proves that

$$AppO_{\mathsf{b}}(AppH_{\mathsf{b}}(h_{\mathsf{a}})).Poss \subseteq \{B\theta\sigma \mid B \longmapsto^{\theta}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi_{\mathsf{hist}} \text{ and } \sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})\}.$$

We need to show the reverse inclusion. For this purpose, suppose B_0 belongs to the right-hand-side of the above inclusion, that is, B_0 has the form $B\theta\sigma$, $B \longmapsto^{\theta}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi_{\mathsf{hist}}$ and $\sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})$.

This derivation can be reordered by postponing the application of R_{his}'s rules, and can be split into two segments, for some HC, θ_1 and θ_2, as follows:

$$B \longmapsto^{\theta_1}_{R_{\mathsf{sta}}} HC \longmapsto^{\theta_2}_{R_{\mathsf{his}}} \chi_{\mathsf{hist}},$$

where $\theta = \theta_1 \circ \theta_2$. This reordering is possible for two reasons:

1. By a well-known result in logic programming theory, called *independence from the selection rule* [37], we can invert the application of two rules in a derivation, provided that none of the two rules rewrites an atom introduced by the other rule.

2. The atoms in the body of R_{his}'s rules, by definition, never match the head of any rule in R_{sta}. So R_{his}'s rules can be delayed until all the necessary rules of R_{sta} have been applied.

Now the reader can easily verify (with (2) and (3)) that $HC\theta_2\sigma$ belongs to $AppH_{\mathsf{b}}(h_a)$, and hence $B\theta\sigma$ (that equals B_0) belongs to $AppO_{\mathsf{b}}(AppH_{\mathsf{b}}(h_{\mathsf{a}})).Poss$. This completes the proof. ∎

The following example uses the Tank Example to illustrate how states may be approximated.

Example 8.5 *As very little is known about* monitor*, the following possibilities must be taken into account:*

- *The* monitor *agent may store in its state any data obtained from other agents (this doesn't mean that* monitor *actually stores all such data);*

- *The* monitor *agent may keep data in its state for unbounded amounts of time (i.e., it cannot be said a priori whether a particular piece of data will be removed or replaced at some point).*

This means that the monitor *agent's state may possibly contain any fact received from other agents. This can be expressed via the following rule:*

$$(\mathbf{r5})\quad \mathcal{B}_{\mathsf{tank1}}(\mathtt{b},\mathtt{F}) \leftarrow \mathtt{told}(\mathtt{X},\mathsf{monitor},\mathtt{F},\mathtt{T}).$$

Clearly, if more information about monitor *is available, the body of the above rule might be enriched with further constraints. For example, by adding* $\mathtt{T} \geq \mathtt{now} - 30$ *one could say that* monitor *does not keep facts for more than 30 minutes. By adding* $\mathtt{X} \neq \mathsf{c}$ *one could say that* c*'s messages are not stored by* monitor.

If R_{his} *consists of rules* (**r1**)-(**r4**), *and* R_{sta} *contains only* (**r5**), *then the condition*

$$B = \mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{L}, \mathsf{tank1} : \boldsymbol{fuel\_level}(\mathtt{T}))) \;\&\; \mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{R}, \mathsf{tank1} : \boldsymbol{region}(\mathtt{T})))$$

has three derivations $B \longmapsto^{\theta_i}_{R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi^i_{\mathsf{hist}}$ $(i = 1,2,3)$, *where* χ^i_{hist} *and* θ_i *are as in Example 8.4 (the first two steps of these derivations apply* (**r5**) *twice, and transform* B *into the constraints HC of Example 8.4; the rest of the derivations coincide with those of Example 8.4).*

The intuitive meaning of these derivations is: two facts approximated by $\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \boldsymbol{fuel\_level}(\mathtt{T}))$ *and* $\mathbf{in}(\mathtt{R}, \mathsf{tank1} : \boldsymbol{region}(\mathtt{T}))$ *may be simultaneously stored in the* monitor *agent's current state when any of the conditions* χ^i_{hist} *is satisfied. For instance,* χ^1_{hist} *is satisfied whenever there exist* X, Y, T′, T″, *such that*

$$\mathtt{X} \neq \mathsf{tank1} \;\&\; \mathtt{T} \leq \mathtt{T}' \;\&\; \mathtt{Y} \neq \mathsf{tank1} \;\&\; \mathtt{T} \leq \mathtt{T}''.$$

This is always possible, whenever there exists an agent different from tank1 *and* monitor*; under this assumption, our rules say that the facts (corresponding to)* $\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \boldsymbol{fuel\_level}(\mathtt{T}))$ *and* $\mathbf{in}(\mathtt{R}, \mathsf{tank1} : \boldsymbol{region}(\mathtt{T}))$ *may be part of the* monitor *agent's current state.*

8.2.3 Consequence Approximation Language

In this section, we show how the agent developer may specify how agent a overestimates agent b's consequence operation. He does so by writing a *consequence approximation program* defined below.

Definition 8.9 (Consequence Approximation Program) *A consequence approximation program used by agent* a *to overestimate agent* b*'s consequence operation is a finite set of rules of the form*

$$\mathcal{B}_{\mathsf{a}}(\mathsf{b}, f) \leftarrow B_1 \,\&\ldots\&B_n,$$

where each B_i *is either a "belief atom" of the form* $\mathcal{B}_{\mathsf{a}}(\mathsf{b},\ldots)$ *or a comparison constraint* T_1 *Op* T_2.

When the developer of agent $\mathfrak{a}$ writes a consequence approximation program R_{con}, then he or she implicitly specifies a consequence operation as shown below:

$$OCn_{\mathfrak{b}}(C) =_{def} \{\text{facts}(B\theta\sigma) \mid B \longmapsto^{\theta}_{R_{\mathsf{con}}} C',\ \sigma \in \mathsf{Sol}(\text{comc}(C')) \text{ and facts}(C')\sigma \subseteq C\}\,. \tag{5}$$

where comc(C') is the set of comparison constraints in C' and facts(C') is the set of facts occurring within the belief atoms of C'.

The following example uses the Tank example to illustrate the concept of a consequence approximation program.

Example 8.6 *Let us make an additional assumption about the* monitor *agent. Suppose we cannot excluded the possibility that the* monitor *agent may derive the current location,* P_{now}*, of* tank1*, from recent information about* tank1*'s low fuel levels and from the region in which* tank1 *is located. This is based on the assumption that if* tank1 *is low in fuel, it must be at the support system located in its region and will stay there for a very short time period (e.g., less than 10 minutes). Hence if* $t < \mathtt{now} - 10$*, then we can safely assume that* monitor *cannot derive* P_{now} *from the region and from* tank1*'s being low in fuel. Then* R_{con} *consists of the following rule:*

$$(\mathbf{r6})\quad \mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{P_{now}}, \mathsf{tank1} : \textit{location}(\mathtt{now}))) \leftarrow C'$$

where C' *is*

$$\begin{aligned}
&\mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{L}, \mathsf{tank1} : \textit{fuel\_level}(\mathtt{T})))\ \ \&\\
&\mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{R}, \mathsf{tank1} : \textit{region}(\mathtt{T})))\ \ \&\\
&\mathtt{T} \geq \mathtt{now} - 10\ \ \&\ \ \mathtt{L} = \mathtt{low}\,.
\end{aligned}$$

Intuitively, C' *means that* tank1 *believes that the region it is in and its being low in fuel at time* $\mathtt{T} \geq \mathtt{now} - 10$ *may be stored in* monitor*'s state at some point. Under this assumption, it is estimated that the* monitor *agent may derive* $\mathbf{in}(\mathtt{P_{now}}, \mathsf{tank1} : \textit{location}(\mathtt{now}))$*, (i.e.* tank1*'s current location), due to the following points:*

- $\mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{P_{now}}, \mathsf{tank1} : \textit{location}(\mathtt{now}))) \longmapsto^{\epsilon}_{R_{\mathsf{con}}} C'$*, where* ϵ *is the empty substitution (the derivation consists of one application of* $(\mathbf{r6})$*);*
- comc$(C') = \{\mathtt{T} \geq \mathtt{now} - 10, \mathtt{L} = \mathtt{low}\}$*;*
- *let* t_0 *be any number such that* $t_0 \geq \mathtt{now} - 10$*; let* $\sigma =_{def} [t_0/\mathtt{T}, low/L]$*; note that* $\sigma \in \mathsf{Sol}(\text{comc}(C'))$*;*
- facts$(\mathcal{B}_{\mathsf{tank1}}(\mathsf{monitor}, \mathbf{in}(\mathtt{P_{now}}, \mathsf{tank1} : \textit{location}(\mathtt{now})))) = \mathbf{in}(\mathtt{P_{now}}, \mathsf{tank1} : \textit{location}(\mathtt{now}))$, *and*
- facts$(C') = \{\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \textit{fuel_level}(\mathtt{T})), \mathbf{in}(\mathtt{R}, \mathsf{tank1} : \textit{region}(\mathtt{T}))\}$

and hence:

$$\begin{aligned}
&\mathbf{in}(\mathtt{P_{now}}, \mathsf{tank1} : \textit{location}(\mathtt{now})) \in\\
&OCn^{\mathsf{tank1}}_{\mathsf{monitor}}(\{\mathbf{in}(\mathtt{low}, \mathsf{tank1} : \textit{fuel\_level}(\mathtt{t_0})), \mathbf{in}(\mathtt{R}, \mathsf{tank1} : \textit{region}(\mathtt{t_0}))\})\,.
\end{aligned}$$

8.2.4 Approximate Secrets Language

As in the previous cases, for the developer of agent $\mathfrak{a}$ to approximate the secrets to be kept from agent $\mathfrak{b}$, he writes a set of rules as described in the following definition.

Definition 8.10 (Approximate Secrets Program) *An* approximate secrets program *used by agent* $\mathfrak{a}$ *to specify secrets to be kept from* $\mathfrak{b}$ *is a finite set of rules of the form*

$$\texttt{secret}_{\mathfrak{a}}(\mathfrak{b}, f) \leftarrow \chi_{\mathsf{cmp}},$$

where f is an approximate fact from $App\mathcal{L}_{\mathfrak{b}}$, and χ_{cmp} is a set of comparison constraints T_1 Op T_2.

Intuitively, the above rule means that f should be kept secret from $\mathfrak{b}$ if χ_{cmp} is true. Every approximate secrets program, R_{sec}, implicitly specifies an abstract secrets function $AppSec(\mathfrak{b})$ as follows:

$$AppSec(\mathfrak{b}) =_{def} \{f\sigma \mid (\texttt{secret}_{\mathfrak{a}}(\mathfrak{b}, f) \leftarrow \chi_{\mathsf{cmp}}) \in R_{\mathsf{sec}} \text{ and } \sigma \in \mathsf{Sol}(\chi_{\mathsf{cmp}})\}. \qquad (6)$$

The Tank example may be used to illustrate the concept of an approximate secrets program.

Example 8.7 *In the Tank example, there is one secret, declared by the following rule:*

$$(\mathbf{r7}) \quad \texttt{secret}_{\mathfrak{tank1}}(\mathfrak{monitor}, \mathbf{in}(\texttt{P}, \mathfrak{tank1} : \boldsymbol{location}(\texttt{T}))) \leftarrow \texttt{T} = \texttt{now}.$$

This means that tank1 *does not want* monitor *to know* tank1*'s current position.*

8.2.5 Agent Approximation Program

Thus, the approximation of $\mathfrak{b}$ used by agent $\mathfrak{a}$ consists of a set of approximation programs as defined above that we collectively call the agent approximation program of $\mathfrak{b}$ used by $\mathfrak{a}$. The following definition collects in one concept the components introduced in the preceding sections.

Definition 8.11 (Agent Approximation Program, $\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}$) *The agent approximation program $\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}$ is a set of rules with the following possible forms:*

history approximation rules $PHC \leftarrow \chi_{\mathsf{hist}}$ *;*

state approximation rules $\mathcal{B}_{\mathfrak{a}}(\mathfrak{b}, f) \leftarrow HC$ *;*

consequence approximation rules $\mathcal{B}_{\mathfrak{a}}(\mathfrak{b}, f) \leftarrow B_1 \& \dots \& B_n$ *;*

secrets approximation rules $\texttt{secret}_{\mathfrak{a}}(\mathfrak{b}, f) \leftarrow \chi_{\mathsf{cmp}}$ *;*

where $f \in App\mathcal{L}_{\mathfrak{b}}$, PHC is a pure history constraint , χ_{hist} is a history code call condition, HC is a set of history constraints , each B_i is either a belief atom of the form $\mathcal{B}_{\mathfrak{a}}(\mathfrak{b}, \dots)$ or a comparison constraint T_1 Op T_2, and χ_{cmp} is a set of comparison constraints.

9 Algorithms for Security Maintenance

In this section, we will define algorithms to compile agent approximation programs, and we will also provide algorithms to perform *static* security checks, as well as *dynamic* security checks. We will focus on algorithms for maintaining data security. Techniques for maintaining action security in *IMPACT* can be found in [9] and [53, Section 10.5.4].

Before proceeding any further, however, we present a result below that shows that if the current history of agent $\mathfrak{a}$ (which $\mathfrak{a}$ surely knows!) is $h_{\mathfrak{a}}$, then the set of secrets violated by agent $\mathfrak{b}$ given that history $h_{\mathfrak{a}}$ has occurred can be precisely characterized in terms of the derivations from $\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}$.

Theorem 9.1 (Violated Secrets As Computations From $\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}$) *Let* χ_{hist} *range over history conditions. Then*

$$\begin{aligned}\mathsf{OViol}_{\mathfrak{b}}(h_{\mathfrak{a}}) \quad = \quad & \{f\theta\sigma \mid (\mathtt{secret}_{\mathfrak{a}}(\mathfrak{b},f) \leftarrow \chi_{\mathsf{cmp}}) \in \mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}, \\ & \boldsymbol{\mathcal{B}}_{\mathfrak{a}}(\mathfrak{b},f)\,\&\chi_{\mathsf{cmp}} \longmapsto^{\theta}_{\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}} \chi_{\mathsf{hist}},\ \sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}}), \\ & \textit{some } \mathbf{in}(E, \mathit{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathfrak{a},\mathfrak{b},\ldots)) \textit{ belongs to } \chi_{\mathsf{hist}}\sigma \textit{ and} \\ & E \textit{ is the last event of } \mathit{Hist}_{\mathfrak{a}}\}\,.\end{aligned}$$

Proof: Let $f_0 \in \mathit{App}\mathcal{L}_{\mathfrak{b}}$ be an arbitrary approximate fact. By definition, $f_0 \in \mathsf{OViol}_{\mathfrak{b}}(h_a)$ iff $f_0 \in \bigcup\{OCn_{\mathfrak{b}}(C) \mid C \in \mathit{App}\mathcal{O}_{\mathfrak{b}}(\mathit{App}H_{\mathfrak{b}}(h_a)).\mathit{New}\}$ and $f_0 \in \mathit{AppSec}(\mathfrak{b})$.

By analogy with the proof of Proposition 8.1, the reader may easily verify (using equations (5) and (4)) that f_0 belongs to some of the above sets $OCn_{\mathfrak{b}}(C)$ iff f_0 has the form $f\theta\sigma$ and

1. $\boldsymbol{\mathcal{B}}_{\mathfrak{a}}(\mathfrak{b},f) \longmapsto^{\theta}_{R_{\mathsf{con}}\cup R_{\mathsf{sta}}\cup R_{\mathsf{his}}} \chi_{\mathsf{hist}}$ with $\sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})$;
2. there exists a code call condition $\mathbf{in}(E, \mathsf{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathfrak{a},\mathfrak{b},\ldots))$ in $\chi_{\mathsf{hist}}\sigma$ such that E is the last event of $\mathsf{Hist}_{\mathfrak{a}}$.

Moreover, by (6), f_0 belongs to $\mathit{AppSec}(\mathfrak{b})$ iff f_0 has the form $f'\sigma'$ and R_{sec} contains a rule

$$\mathtt{secret}_{\mathfrak{a}}(\mathfrak{b},f) \leftarrow \chi_{\mathsf{cmp}}$$

such that $\sigma' \in \mathsf{Sol}(\chi_{\mathsf{cmp}})$. As a consequence of 1) and 2), we obtain the two points below:

a) Assume $f_0 \in \mathsf{OViol}_{\mathfrak{b}}$. Then, since $\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}} \supseteq R_{\mathsf{con}} \cup R_{\mathsf{sta}} \cup R_{\mathsf{his}}$, the derivation in 1) is also a derivation $\boldsymbol{\mathcal{B}}_{\mathfrak{a}}(\mathfrak{b},f) \longmapsto^{\theta}_{\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}} \chi_{\mathsf{hist}}\sigma$. Consider a ground instance $\boldsymbol{\mathcal{B}}_{\mathfrak{a}}(\mathfrak{b},f_0) \longmapsto^{\epsilon}_{\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}} \chi_{\mathsf{hist}}\sigma$ of the above derivation. It can be immediately extended to $\boldsymbol{\mathcal{B}}_{\mathfrak{a}}(\mathfrak{b},f_0)\ \&\ \chi_{\mathsf{cmp}}\sigma' \longmapsto^{\epsilon}_{\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}} \chi_{\mathsf{hist}}\sigma\ \&\ \chi_{\mathsf{cmp}}\sigma'$, by appending $\&\ \chi_{\mathsf{cmp}}\sigma'$ to each goal. Now, note that the empty substitution ϵ is in $\mathsf{Sol}(\chi_{\mathsf{hist}}\sigma\ \&\ \chi_{\mathsf{cmp}}\sigma')$, and that $\chi_{\mathsf{hist}}\sigma\ \&\ \chi_{\mathsf{cmp}}\sigma'$ contains a code call condition $\mathbf{in}(E, \mathsf{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathfrak{a},\mathfrak{b},\ldots))$ such that E is the last event of $\mathsf{Hist}_{\mathfrak{a}}$.

   By a standard logic programming result ([37, Lifting Lemma]), this derivation can be "lifted" to a derivation $\boldsymbol{\mathcal{B}}_{\mathfrak{a}}(\mathfrak{b},f)\ \&\ \chi_{\mathsf{cmp}} \longmapsto^{\theta'}_{\mathbf{AAP}^{\mathfrak{a}}_{\mathfrak{b}}} \chi'_{\mathsf{hist}}$. Clearly, χ'_{hist} has a solution σ'' such that $\chi'_{\mathsf{hist}}\sigma''$ contains a code call condition $\mathbf{in}(E, \mathsf{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathfrak{a},\mathfrak{b},\ldots))$ where E is the last event of $\mathsf{Hist}_{\mathfrak{a}}$. This proves that f_0 belongs to the right-hand-side of the equation in this theorem's statement.

b) Conversely, suppose that f_0 belongs to the right-hand-side of the equation in the theorem's statement. Then we have $\mathcal{B}_{\mathtt{a}}(\mathtt{b}, f) \,\&\, \chi_{\mathsf{cmp}} \longmapsto^{\theta}_{\mathbf{AAP}^{\mathtt{a}}_{\mathtt{b}}} \chi_{\mathsf{hist}}$, $\sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})$, and for some call $\mathbf{in}(E, \mathsf{Hist}_{\mathtt{a}} : \mathit{retrieve\_answ}(\mathtt{a}, \mathtt{b}, \ldots))$ in $\chi_{\mathsf{hist}}\sigma$, E is the last event of $\mathsf{Hist}_{\mathtt{a}}$. From this derivation, by dropping the part corresponding to χ_{cmp} from each goal, we obtain a derivation $\mathcal{B}_{\mathtt{a}}(\mathtt{b}, f) \longmapsto^{\theta}_{\mathbf{AAP}^{\mathtt{a}}_{\mathtt{b}}} \chi'_{\mathsf{hist}}$, where $\chi'_{\mathsf{hist}}\sigma$ still contains the above code call condition (the part removed from χ_{hist} consists only of pure comparison constraints). The above derivation cannot use rules from R_{sec} (which match neither the initial goal nor the bodies of $\mathbf{AAP}^{\mathtt{a}}_{\mathtt{b}} - R_{\mathsf{sec}}$); therefore, it is also a derivation $\mathcal{B}_{\mathtt{a}}(\mathtt{b}, f) \longmapsto^{\theta}_{R_{\mathsf{con}} \cup R_{\mathsf{sta}} \cup R_{\mathsf{his}}} \chi'_{\mathsf{hist}}$.

It follows by 1) and 2) that $f_0 \in \bigcup\{\mathit{OCn}_{\mathtt{b}}(C) \mid C \in \mathit{App}\mathcal{O}_{\mathtt{b}}(\mathit{AppH}_{\mathtt{b}}(h_a)).\mathit{New}\}$. Moreover, note that χ_{hist} contains $\chi_{\mathsf{cmp}}\theta$ and σ is a solution of χ_{hist}, so $\theta\sigma$ is a solution to χ_{cmp}. It follows, by (6), that $f\theta\sigma$ – that is, f_0 – is in $\mathit{AppSec}(\mathtt{b})$.

We may conclude that $f_0 \in \mathsf{OViol}_{\mathtt{b}}(h_a)$.

From a) and b) we immediately derive that f_0 belongs to the left-hand-side of the equation in the theorem's statement iff it belongs to the right-hand-side. This completes the proof. ∎

The following example shows how this theorem may be used to determine which secrets are violated by a given agent b w.r.t. a given history.

Example 9.1 *In our example,* $\mathbf{AAP}^{\mathtt{tank1}}_{\mathtt{monitor}}$ *consists of rules* (**r1**)-(**r7**). *The unique secret is specified by* (**r7**), *thus the security check is only concerned with derivations starting from the corresponding condition* $G_0 = \mathcal{B}_{\mathtt{tank1}}(\mathtt{monitor}, \mathbf{in}(\mathtt{P}, \mathtt{tank1} : \mathit{location}(\mathtt{T}))) \,\&\, \mathtt{T} = \mathtt{now}$. *Only one such derivation reaches a history condition* χ_{hist} *that mentions* $\mathit{Hist}_{\mathtt{tank1}}$. *This derivation uses rules* (**r6**),(**r5**),(**r5**),(**r3**),(**r1**), *and yields a condition of the form*

$$\begin{aligned}\chi_{\mathsf{hist}} \;=\; & \mathbf{in}(\mathtt{Ev}_4, \mathit{Hist}_{\mathtt{tank1}} : \mathit{retrieve\_answ}(\mathtt{tank1}, \mathtt{monitor}, \mathbf{in}(\mathtt{L1}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T1})), \_)) \;\& \\ & L1 = \mathit{low} \;\& \\ & \mathtt{T3}_4 \geq \mathtt{Ev}_4.\mathtt{time} \;\& \\ & \mathtt{Y}_5 \neq \mathtt{monitor} \;\&\; \mathtt{T}_1 < \mathtt{T2}_5 \;\& \\ & \mathtt{T}_1 \geq \mathtt{now} - 10 \;\& \\ & \mathtt{T} = \mathtt{now}\,.\end{aligned}$$

After evaluating the above code call to $\mathit{Hist}_{\mathtt{tank1}}$, *one can always set* $\mathtt{T3}_4 := \mathtt{Ev}_4.\mathtt{time}$, $\mathtt{L1} = \mathtt{low}$, $\mathtt{Y}_5 := \mathtt{c}$, $\mathtt{T2}_5 := \mathtt{T}_1 + 1$, *and* $\mathtt{T} := \mathtt{now}$. *Subsequently, the only constraint that might not be satisfied is* $\mathtt{T}_1 \geq \mathtt{now} - 10$. *Therefore,* χ_{hist} *has a solution if and only if the code call retrieve_ans finds an answer message from* tank1 *to* monitor *containing a fact* $\mathbf{in}(\mathtt{low}, \mathtt{tank1} : \mathit{fuel\_level}(\mathtt{T}))$ *where* $\mathtt{T}_1 \geq \mathtt{now} - 10$.

Data security, however, is violated only if the answer message found by retrieve_answ is the last message of $\mathit{Hist}_{\mathtt{tank1}}$.

Intuitively, all this means is that if tank1 *tries to send the* monitor *agent information about its being low on fuel during the last 10 minutes, then a security violation is detected. A closer examination of the rules used in the derivation reveals that* $\mathbf{AAP}^{\mathtt{tank1}}_{\mathtt{monitor}}$ *"discovers" that* monitor *might combine the fact that* tank1 *is low in fuel with its region coming from another agent* $\mathtt{Y}_5 \neq \mathtt{tank1}$, *and derive* tank1*'s current position.*

The following function compiles $\mathbf{AAP}_{\mathsf{b}}^{\mathsf{a}}$ into a set of tuples of the form

$$\langle \mathsf{b}, f, \chi_{\mathsf{hist}} \rangle$$

where b is an agent name, f is an approximate fact from $App\mathcal{L}_{\mathsf{b}}$, and χ_{hist} is a history condition. The intended meaning of the above tuple is that for all $\sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})$, $f\sigma$ belongs to $\mathsf{OViol}_{\mathsf{b}}(h_{\mathsf{a}})$. We use the notation **OVT** to denote this set of tuples and call the table, the *overestimated violation table.* The following definition provides a method to compile the above table.

Definition 9.1 (Compilation) *Function* $\mathsf{SecP}_{\mathsf{a}} : CompileAAP(\mathbf{AAP}_{\mathsf{b}}^{\mathsf{a}})$ *sets* **OVT** *to the set of all tuples* $\langle \mathsf{b}, f, \chi_{\mathsf{hist}} \rangle$ *such that:*

1. $(\mathtt{secret}_{\mathsf{a}}(\mathsf{b}, f) \leftarrow \chi_{\mathsf{cmp}}) \in \mathbf{AAP}_{\mathsf{b}}^{\mathsf{a}}$*;*
2. $(\boldsymbol{\mathcal{B}}_{\mathsf{a}}(\mathsf{b}, f)\ \&\chi_{\mathsf{cmp}}) \overset{\mathsf{mg}\ \theta}{\longmapsto}_{\mathbf{AAP}_{\mathsf{b}}^{\mathsf{a}}} \chi_{\mathsf{hist}}$ *and* χ_{hist} *is a history condition;*
3. *the set of comparison constraints in* χ_{hist} *is satisfiable.*

The following example uses the Tank example to illustrate the compilation procedure.

Example 9.2 *In the Tanks example* $\mathbf{OVT}_{\mathsf{now}}$ *would contain the tuple*

$$\langle \mathsf{monitor}, \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \boldsymbol{location}(\mathtt{now})), \chi_{\mathsf{hist}} \rangle,$$

where χ_{hist} *is the history condition described in Example 9.1. The set of comparison constraints in* χ_{hist} *can be satisfied by setting:* $\mathtt{T3}_4 := \mathtt{Ev}_4.\mathtt{time}$, $\mathtt{L1} = \mathtt{low}$, $\mathtt{Y}_5 := \mathtt{c}$, $\mathtt{T2}_5 := \mathtt{T}_1 + 1$, $\mathtt{T} := \mathtt{now}$, *and* $\mathtt{T}_1 := \mathtt{now} - 9$.

Before continuing to the next section, we note that Step (2) of **CompileAAP** may be performed in polynomial time data complexity by using standard table-based resolution methods such as those implemented in the well-known XSB system (`http://xsb.sourceforge.net`) developed at Stonybrook.

9.1 Dynamic Security Verification Algorithm

Once the table **OVT** is constructed, security may be verified dynamically via a function $\mathsf{SecP}_{\mathsf{a}} : DynOViol(\mathsf{b}, \mathtt{Ans})$, that computes $\mathsf{OViol}_{\mathsf{b}}(h_{\mathsf{a}} \cdot e)$ where e is an event corresponding to a's current answer $\mathtt{Ans}$ to b. The dynamic security check algorithm is given below (Algorithm 9.1).

It is important to note that the dynamic security check algorithm does *not* modify $\mathsf{Hist}_{\mathsf{a}}$ — it merely checks whether some secret would be violated if $\mathtt{Ans}$ were returned to b. The following theorem states that Algorithm 9.1 is correct.

Theorem 9.2 (Correctness of Dynamic Security Check) *Let* **OVT** *be the table constructed by*
$\mathsf{SecP}_{\mathsf{a}} : CompileAAP(\mathbf{AAP}_{\mathsf{b}}^{\mathsf{a}})$*, and let* e *be an answer message from* a *to* b *with answer* $\mathtt{Ans}$*. Then*

$$\mathsf{OViol}_{\mathsf{b}}(h_{\mathsf{a}} \cdot e) = \mathsf{SecP}_{\mathsf{a}} : DynOViol(\mathsf{b}, \mathtt{Ans})\,.$$

```
Algorithm 9.1 (Dynamic Security Check)
SecP_a : DynOViol(b : AgentName, Ans : Answer)

(⋆ output: an overestimation $\mathbf{OVT}_{\mathsf{now}}$ of the set of secrets ⋆)
(⋆ that would be violated if Ans were returned to b ⋆)

  $\mathbf{OVT}_{\mathsf{now}} := \emptyset$;
  (⋆ $Hist_{\mathtt{a}}$ is temporarily extended with answer message e ⋆)
  $e := \mathbf{new}(AnswerMessage)$;
  e.sender := a;
  e.receiver := b;
  e.answer := Ans;
  e.SendTime := now;
  execute insert_answ(e);
  (⋆ OVT's tuples are evaluated against the extended history ⋆)
  for all tuples $\langle \mathtt{b}, f, \chi_{\mathsf{hist}}\rangle$ in OVT do
        for all $\sigma$ in $\mathsf{Sol}(\chi_{\mathsf{hist}})$ do
              for all $\mathbf{in}(V, Hist_{\mathtt{a}} : retrieve\_answ(\ldots))$ in $\chi_{\mathsf{hist}}\sigma$ do
                    if $V = e$ then $\mathbf{OVT}_{\mathsf{now}} := \mathbf{OVT}_{\mathsf{now}} \cup \{f\sigma\}$;
  (⋆ $Hist_{\mathtt{a}}$ is restored ⋆)
  execute delete(e);
  return($\mathbf{OVT}_{\mathsf{now}}$);
end.
```

Proof: By Theorem 9.1 and Definition 9.1, an approximate fact f_0 is in $\mathsf{OViol}_{\mathtt{b}}(h_a)$ iff there exist a triple $\langle \mathtt{b}, f, \chi_{\mathsf{hist}}\rangle$ in **OVT** and a substitution $\sigma \in \mathsf{Sol}(\chi_{\mathsf{hist}})$ such that

1. $f\sigma = f_0$;
2. $\chi_{\mathsf{hist}}\sigma$ contains some code call condition $\mathbf{in}(E, \mathsf{Hist}_{\mathtt{a}} : retrieve\_answ(\mathtt{a}, \mathtt{b}, \ldots))$ where E is the last event of $\mathsf{Hist}_{\mathtt{a}}$.

Now, Algorithm 9.1 clearly returns all and only the $f\sigma$ satisfying the above properties. The theorem follows immediately. ∎

We say that **OVT** is *bounded* iff there is an integer k such that for every triple $\langle \mathtt{b}, f, \chi_{\mathsf{hist}}\rangle$ in **OVT**, χ_{hist} contains at most k variables in it. When **OVT** is bounded, it is now easy to see that the dynamic security check algorithm above is polynomial in the size of the history and the size of **OVT**. Boundedness is a condition satisfied in most practical applications — after all we rarely need to execute code call conditions with more than (say) 100 variables in it.

9.2 Static/Combined Security Verification Algorithm

The dynamic security verification algorithm defined in the preceding section performs a polynomial run time test that agent a must execute whenever another agent b makes a request. In contrast, static security verification tries to ensure *prior* to deploying an agent, that the agent's way of answering queries is secure independently of the histories that actually arise over time. In order to implement static security, the developer of an agent $\mathfrak{a}$ must specify an overestimate $\text{pos}\mathcal{H}_{\mathfrak{a}}^{+}$ of histories that $\mathfrak{a}$ may participate in the future. This can be done via a set of rules that the agent developer must write.

Definition 9.2 (Self approximation Program) *Agent $\mathfrak{a}$'s* self approximation program *is a finite set R_{slf} of rules having the form*

$$\mathbf{in}(e, \mathit{Hist}_{\mathfrak{a}} : \mathit{fun}(\mathtt{args})) \leftarrow \chi_{\mathsf{cmp}},$$

where fun is one of the functions of package $\mathit{Hist}_{\mathfrak{a}}$, args is a suitable list of arguments, and χ_{cmp} is a comparison constraint.

Intuitively, the rules of R_{slf} are used jointly with the rules in the agent approximation program $\mathbf{AAP}_{\mathfrak{b}}^{\mathfrak{a}}$ to derive a set of comparison constraints χ_{cmp} by iteratively performing derivations. If any such χ_{cmp} is satisfiable, then a security violation may occur. Before proceeding to define the static security verification algorithm, we first present an intermediate definition.

Definition 9.3 ($\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$) *Suppose χ_{hist} is a set of history conditions, and*

$$\chi_0 = \mathbf{in}(e', \mathit{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathtt{args}))\,.$$

Then we use $\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$ to denote the set of history conditions obtained from χ_{hist} by adding the constraints:

- $e.\mathtt{time} \leq \mathtt{now}$ *for each code call of the form $\mathbf{in}(e, \mathit{Hist}_{\mathfrak{a}} : \mathit{fun}(\ldots, w))$ in χ_{hist},*
- *if w has the form $\mathit{Op}t$, then $e.\mathtt{time}\ \mathit{Op}\ t$ is added to χ_{hist} where $\mathbf{in}(e, \mathit{Hist}_{\mathfrak{a}} : \mathit{fun}(\ldots, w))$ is in χ_{hist}, $\mathtt{e.time} = \mathtt{now}$ to χ_{hist},*
- $e.\mathtt{time} = \mathtt{now}$ *is added to χ_{hist} for a selected code call condition of the form $\mathbf{in}(e', \mathit{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathtt{args}))$ in χ_{hist}.*

Note that the last condition above will be true iff e' is the last event in $\mathsf{Hist}_{\mathfrak{a}}$. It is important to note that depending upon which $\chi_0 = \mathbf{in}(e', \mathsf{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathtt{args}))$ is selected from χ_{hist}, the definition of $\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$ changes — hence, we use the notation $\mathsf{EXT}_{\mathtt{now}}(\chi_{\mathsf{hist}})$ to denote the set of all $\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$ for χ_0 in χ_{hist} having the form $\mathbf{in}(e', \mathsf{Hist}_{\mathfrak{a}} : \mathit{retrieve\_answ}(\mathtt{args}))$. The following example shows the construction of $\mathsf{EXT}_{\mathtt{now}}(\chi_{\mathsf{hist}})$.

Example 9.3 *Consider the χ_{hist} of the only tuple in the $\mathbf{OVT}_{\mathsf{now}}$ computed in Example 9.2. It contains one call to $\mathit{Hist}_{\mathfrak{tank1}}$, namely,*

$$\chi_0 =_{def} \mathbf{in}(\mathtt{Ev}_4, \mathit{Hist}_{\mathfrak{tank1}} : \mathit{retrieve\_answ}(\mathfrak{tank1}, \mathfrak{monitor}, \mathbf{in}(\mathtt{L1}, \mathfrak{tank1} : \boldsymbol{\mathit{fuel\_level}}(\mathtt{T1})), \_))$$

Thus, the extended condition in this example is:

$$\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0) = \chi_{\mathsf{hist}}\ \&\ \mathtt{Ev_4.time} \leq \mathtt{now}\ \&\ \mathtt{Ev_4.time} = \mathtt{now}\,.$$

If the last parameter of retrieve_answ were—say—"$> \mathtt{T_9}$", then $\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$ would contain also a constraint $\mathtt{Ev_4.time} > \mathtt{T_9}$.

```
Algorithm 9.2 (Static Security Check) $\mathit{SecP}_{\mathsf{a}} : \mathit{StaticOViol}(\mathsf{b} : \mathtt{AgentName})$

(⋆ output: a modified table $\mathbf{OVT}_{\mathsf{opt}}$ ⋆)

    $\mathbf{OVT}_{\mathsf{opt}} := \emptyset$;
    (⋆ $\mathbf{OVT}$'s tuples are evaluated using $R_{\mathsf{slf}}$ ⋆)
    for all tuples $\langle \mathsf{b}, f, \chi_{\mathsf{hist}} \rangle$ in $\mathbf{OVT}$ do
            for all $\chi'_{\mathsf{hist}} \in \mathsf{EXT}_{\mathtt{now}}(\chi_{\mathsf{hist}})$ do
                    for all deriv. $\chi'_{\mathsf{hist}} \overset{\mathsf{mg}\ \theta}{\longmapsto}_{R_{\mathsf{slf}}} \chi_{\mathsf{cmp}}$ such that $\chi_{\mathsf{cmp}}$ is a comparison constraint
                            do if $\mathsf{Sol}(\chi_{\mathsf{cmp}}) \neq \emptyset$ then $\mathbf{OVT}_{\mathsf{opt}} := \mathbf{OVT}_{\mathsf{opt}} \cup \{\langle \mathsf{b}, f, \chi_{\mathsf{hist}} \rangle\}$ ;
    return($\mathbf{OVT}_{\mathsf{opt}}$);
end.
```

We are now ready to specify the algorithm for static security checks. As mentioned earlier, this function extends the derivations from $\mathbf{AAP}^{\mathsf{a}}_{\mathsf{b}}$ with derivations from R_{slf}, until a set of comparison constraints χ_{cmp} is obtained. If χ_{cmp} is satisfiable, then a security violation may occur. In practice, the algorithm uses the precomputed derivations stored in **OVT**, and computes only the derivations from R_{slf}. It returns a modified violation table $\mathbf{OVT}_{\mathsf{opt}}$ corresponding to possible security violations.

The intuition is that if a tuple of the form $\langle \mathsf{b}, f, \chi_{\mathsf{hist}} \rangle$ is in $\mathbf{OVT}_{\mathsf{opt}}$, then χ_{hist} might become true at some future point in time (according to R_{slf}), and in that case, b might violate f. In other words, the static security check coincides with ensuring that

$$\mathit{SecP}_{\mathsf{a}} : \mathit{StaticOViol}(\mathsf{b}) = \emptyset\,.$$

The following example revisits the Tank Example and illustrates the use of the static security algorithm.

Example 9.4 *Consider two possible cases. In the first case, the* $\mathsf{tank1}$ *agent does not provide information on its fuel level in the last 10 minutes to the* $\mathsf{monitor}$ *agent. In this case we will show that* $\mathsf{tank1}$ *is statically secure. In the second scenario* $\mathsf{tank1}$ *may tell the* $\mathsf{monitor}$ *agent its fuel level in the last 7 minutes. We will show that in this case* $\mathsf{tank1}$ *may indirectly disclose a secret.*

Case 1 *In this implementation, all answers of the form* $\mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T}))$ *satisfy* $\mathtt{T} \leq \mathtt{now} - \mathtt{11}$. *This can be expressed by the following self-approximation rule:*

$$(\mathbf{r8}) \quad \mathbf{in}(\mathtt{E}, \mathit{Hist}_{\mathsf{tank1}} : \mathit{retrieve\_answ}(\mathsf{tank1}, \mathsf{monitor}, \mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T})), \mathtt{W})) \leftarrow \mathtt{T} \leq \mathtt{now} - \mathtt{11}\,.$$

Returning to the χ_{hist} *of the only tuple in* $\mathbf{OVT}_{\mathsf{now}}$ *of* $\mathsf{tank1}$. *The extended condition* $\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$ *(see Example 9.3) can be evaluated using* $(\mathbf{r8})$, *which yields the set of con-*

straints

$$\begin{aligned}\chi_{\mathsf{cmp}} \;=\;& \mathtt{T}_1 \leq \mathtt{now} - 11 \;\&\\ & \mathtt{T3}_4 \geq \mathtt{Ev}_4\mathtt{.time} \;\&\\ & \mathtt{Y}_5 \neq \mathsf{tank1} \;\&\; \mathtt{T}_1 < \mathtt{T2}_5 \;\&\\ & \mathtt{T}_1 \geq \mathtt{now} - 10 \;\&\\ & \mathtt{L1} = \mathtt{low} \;\&\\ & \mathtt{T} = \mathtt{now} \;\&\\ & \mathtt{Ev}_4\mathtt{.time} \leq \mathtt{now} \;\&\\ & \mathtt{Ev}_4\mathtt{.time} = \mathtt{now}\,.\end{aligned}$$

The first row comes from (**r8**)*, while the others were already in* $\mathsf{ext}_{\mathtt{now}}(\chi_{\mathsf{hist}}, \chi_0)$*. This set of constraints is not satisfiable because it contains the mutually incompatible constraints* $\mathtt{T}_1 \leq \mathtt{now} - 11$ *and* $\mathtt{T}_1 \geq \mathtt{now} - 10$*. Thus, our static security check proves that providing the fuel level service is secure as far as* monitor *is concerned. We recall the main assumptions (encoded in the approximation rules) that support this result:*

- *agent* monitor *may get all sorts of information from agents different from* tank1*;*
- *The* monitor*'s state may contain any subset (possibly all) of the data obtained from other agents;*
- *The* monitor *may derive* tank1*'s current position from its region and its being low in fuel in the last 10 minutes.*

The security check certifies that under the above conditions, the monitor *agent will never violate* tank1*'s current position* due to tank1's answers*. That is, the derivation involving* $\mathit{Hist}_{\mathsf{tank1}}$ *leads (with* (**r8**)*) to an unsatisfiable conjunction of comparison constraints* χ_{cmp}*. In this case,* $\mathsf{Sol}(\chi_{\mathsf{cmp}}) = \emptyset$ *and hence no tuple is added to* $\mathbf{OVT}_{\mathsf{opt}}$ *(see the above algorithm). The other derivations never mention* $\mathit{Hist}_{\mathsf{tank1}}$*; this implies that* $\mathsf{EXT}_{\mathtt{now}}(\chi_{\mathsf{hist}}) = \emptyset$*; therefore, no tuples of* $\mathbf{OVT}_{\mathsf{opt}}$ *are obtained from such derivations. It follows that*

$$\mathsf{SecP}_{\mathsf{tank1}} : \mathit{StaticOViol}(\mathsf{monitor}) = \emptyset,$$

and hence, tank1 *is statically secure.*

Case 2 *Suppose* R_{slf} *is extended with a corresponding rule*

$$(\mathbf{r8'})\quad \mathbf{in}(\mathtt{E}, \mathit{Hist}_{\mathsf{tank1}} : \mathit{retrieve\_answ}(\mathsf{tank1}, \mathsf{monitor}, \mathbf{in}(\mathtt{L}, \mathsf{tank1} : \mathit{fuel\_level}(\mathtt{T})), \mathtt{W})) \leftarrow \mathtt{T} \leq \mathtt{now} - 7\,.$$

Now there would be another derivation $\chi_{\mathsf{hist}} \longmapsto^{\theta}_{R_{\mathsf{slf}}} \chi'_{\mathsf{cmp}}$ *(where* χ_{hist} *is defined as in the previous case), such that*

$$\begin{aligned}\chi'_{\mathsf{cmp}} &= \mathtt{T}_1 \leq \mathtt{now} - 7 \ \& \\ &\quad \mathtt{T3}_4 \geq \mathtt{Ev}_4.\mathtt{time} \ \& \\ &\quad \mathtt{L} = \mathtt{low} \ \& \\ &\quad \mathtt{Y}_5 \neq \mathsf{tank1} \ \& \ \mathtt{T}_1 < \mathtt{T2}_5 \ \& \\ &\quad \mathtt{T}_1 \geq \mathtt{now} - 10 \ \& \\ &\quad \mathtt{T} = \mathtt{now} \ \& \\ &\quad \mathtt{Ev}_4.\mathtt{time} \leq \mathtt{now} \ \& \\ &\quad \mathtt{Ev}_4.\mathtt{time} = \mathtt{now}\,.\end{aligned}$$

These comparison constraints are satisfiable with any $\mathtt{T}_1$ *such that* $\mathtt{now} - 10 \leq \mathtt{T}_1 \leq \mathtt{now} - 7$, *and hence*

$$\mathsf{SecP}_{\mathsf{tank1}} : \mathit{StaticOViol}(\mathsf{monitor}) = \{\langle \mathsf{monitor}, \mathbf{in}(\mathtt{P}, \mathsf{tank1} : \boldsymbol{\mathtt{location}}(\mathtt{now})), \chi_{\mathsf{hist}} \rangle\}\,.$$

This means that tank1 *may indirectly disclose the secret if condition* χ_{hist} *becomes true at some point.*

In fact, we can combine static and dynamic security verification by: (i) removing all entries from **OVT** whose history conditions will never be satisfied (according to the self-approximation rules R_{slf}). Now, if R_{slf} is correct, then we may replace the table **OVT** by $\mathbf{OVT}_{\mathsf{opt}}$ in the Dynamic Security check algorithm given earlier in the paper. Doing so has the following obvious advantages:

- dynamic security verification becomes more efficient, because less entries have to be considered;
- the resulting histories are in general more cooperative than statically secure histories, because those services which are not guaranteed to be secure at compile time (given the necessarily imprecise predictions about a's future histories) are given another choice at run-time, instead of being restricted a priori.

The following example revisits the Tank Example and illustrates the working of combined security verification.

Example 9.5 *In the scenario of the first case of Example 9.4, the combined check would return an empty table* $\mathbf{OVT}_{\mathsf{opt}}$*; this would automatically turn off run-time verification. The second case of Example 9.4 is less fortunate: there,* $\mathbf{OVT}_{\mathsf{opt}}$ *coincides with* **OVT**, *and no advantage is obtained at run-time. It is possible to find intermediate cases where* $\emptyset \subset \mathbf{OVT}_{\mathsf{opt}} \subset \mathbf{OVT}$.

In this section, we have developed algorithms to perform both *static* security checks, as well as *dynamic* security checks for handling data security. The **CompileAAP** procedure is easily implementable on top of a commercial Prolog system (e.g. XSB) and runs in polynomial time. We define a table **OVT** which overestimates the set of violated secrets. Using this table, the dynamic security algorithm automatically checks security run time. Furthermore, this check is polynomial as long as **OVT**'s size is bounded by some constant.

10 Related Work

Most research on agent security deals with issues related to the usage of agents on the Web. Attempts have been made to answer questions such as, "Is it safe to click on a given hyperlink"? or "If I send this program out into the Web to find some bargain CD's, will it get cheated?" (e.g., [15, 16]). Others try to develop methods for finding intruders who are executing programs not normally executed by "honest" users or agents [17]. In contrast, in this paper, we focus on data security and action security in multi-agent environments.

A significant body of work has also gone into ensuring that agents neither crash their host nor abuse its resources. Most mobile-agent systems protect the hosts by [25]: (1) cryptographically verifying the identity of the agent's owner, (2) assigning access restrictions to the agent based on the owner's identity, and (3) allowing the agent to execute in a secure execution environment that can enforce these restrictions [59]. Java agent security relies mainly on the idea of that an applet's actions are restricted to its "sandbox," an area of the web browser dedicated to that applet [24]. Java developers claim that their Java 2 platform provides both system security and information security [29].

An interesting approach for safe execution of untrusted code is the Proof-Carrying Code (PCC) technique [43]. In a typical instance of PCC, a code receiver establishes a set of safety rules that guarantee safe behavior of programs, and the code producer creates a formal safety proof that proves, for the untrusted code, adherence to the safety rules. Then, the receiver is able to use a simple and fast proof validator to check, with certainty, that the proof is valid and hence the untrusted code is safe to execute. An important advantage of this technique is that although there might be a large amount of effort in establishing and formally proving the safety of the untrusted code, almost the entire burden of doing this is on the code producer. The code consumer, on the other hand, has only to perform a fast, simple, and easy-to-trust proof-checking process.

Campbell and Qian [11] address security issues in a mobile computing environment using a mobile agent based security architecture. This security architecture is capable of supporting dynamic application specific security customization and adaptation. In essence the idea is to embed security functions in mobile agents to enable runtime composition of mobile security systems. The implementation is based on OMG's *CORBA* distributed object orientation technology and Java-based distributed programming environment. Campbell and Qian [11] address security issues in a mobile computing environment using a mobile agent based security architecture. This security architecture is capable of supporting dynamic application specific security customization and adaptation. In essence the idea is to embed security functions in mobile agents to enable runtime composition of mobile security systems. The implementation is based on OMG's *CORBA* distributed object orientation technology and Java-based distributed programming environment. Gray et al. [25] consider a problem of protecting a group of machines which do not belong to the same administrative control. They propose a market-based approach in which agents pay for their resources.

Less attention has been devoted to the opposite problem, that is, protecting mobile agents from their hosts [47]. An example of how to protect Java mobile agents is given in [44]. Hohl [28] proposed to protecting mobile agents from attackers by not giving the attacker enough time to manipulate the data and code of the agent. He proposed that this can be achieved by a combination of a *code mess up* and *limited lifetime of code and data* which he describes. Farmer et al. [22] use a *state appraisal* mechanism which checks if some invariants of the agent's state hold (e.g., relationships among variables) when an agent reaches a new execution environment. Vigna [58] presents a mechanism to detect possible illegal modification of a mobile agent which is based on post-mortem analysis of data–called *traces*—that are collected during agent execution. The traces are used for checking the agent program against a supposed history of execution.

At the same level of abstraction, it is necessary to deal with issues of identity verification and message exchange protection [57]. For example, Thirunavukkarasu et al. proposed an architecture for KQML which is based on cryptographic techniques. It allows agents to verify the identity of other agents, detect message integrity violations, protect the privacy of messages and ensure non-repudiation of message origin. The techniques and methodologies which we presented in this paper rely on the assumption that the above problems and other network security problems [48] are dealt with correctly by the underlying implementation.

Zapf et al. [65] consider security threats to both hosts and agents in electronic markets. They describe the preliminary security facilities implemented in their agent system AMETAS. They do not provide a formal model or an experimental results to evaluate their system.

Agent data security has many analogies with security in databases. This field has been studied intensively, e.g. [5, 6, 10, 12, 31, 41, 62]. While this work is significant, none of it has focused on agents. We attempt to build on top of existing approaches. However, data security in autonomous agents environments raises new problems. In particular, no central authority can maintain security, but rather participants in the environment should be responsible for maintaining it.

In [7], a modal logic of security is defined, based on a modal knowledge operator, $\mathbf{K_A}$, and a modal operator $\mathbf{R_A}$, that captures what the user is allowed to know. The semantics of $\mathbf{R_A}$ and $\mathbf{K_A}$ are similar; both of them satisfy all the axioms of modal logic S5 (i.e. axioms *K, T, 4, 5*, see [14].) Cuppens [18] adapts the logical framework introduced in [7] to study aggregation problems. He provides an elegant logical treatment of data security in which a "diamond"-like (w.r.t. standard modal logic [14]) semantics is provided for $\mathbf{R_A}$. The user is only allowed to have correct beliefs.

There are significant differences between our work and the excellent work of [7, 18]. First, our agents automatically couple actions to state changes. In their case, they do not consider agents, and hence, the possibility of action security does not arise. Additionally, the fact that data security can be violated because of action security is not an issue. Second, we provide various important "approximation" notions that they do not provide. Third, our security specification languages are new and interesting as are our static algorithm for security maintenance runs in polynomial time, and our dynamic algorithm also runs in polynomial time as long as the boundedness condition is satisfied.

Problems of authentication and authorization arise when databases operate in an open environment [8]. Bina et al. propose a framework for solving these problems using WWW information servers and a modified version of the NCSA Mosaic. Berkovits et al. [4] consider this problem in mobile agent systems by modeling the trust relation between the principals of the mobile agents. We do not consider the authentication problem in our work, but rather assume that methods such as developed in [8] are available. Usually these methods used cryptography and electronic signatures techniques. A tutorial text on such techniques can be found in [51].

Formal models for verifying security of protocols for authentication, key distribution, or information sharing may have some similarities with our formal model. Heintze and Tygar [27] present a simple model which includes the notions of traces (similar to our histories), agent states and beliefs. Our notions are more general than theirs. For example, the internal state of each agent in their model consists of three components: (1) the set of messages and keys known to the agent; (2) the set of messages and keys believed by the agent to be secret (and with whom the secrets are shared); and (3) the set of nonces recently generated by the agent. We do not make any restrictions on the agents' states and we assume that an agent can infer new information from its beliefs.

We also define the notion of approximating agent security and provide a language within which the developer of an agent can express the approximations that its agent must use. Their system is used to verify the security of cryptographic protocols. They present an interesting result concerning a composition of two secure protocols. They state sufficient conditions on two secure protocols A and B such that they may be combined to form a new secure protocol C.

In some systems agents are used to maintain security. For example, in the architecture presented in [3] of Java-based agents for information retrieval, there are two security agents: Message Security Agent (MSA) and Controller Security Agents (CSA). The MSA deals with services relating to the exchange of messages. The CSA provides services to check adequate use of resources by detecting anomalies. We do not consider the basic security problems provided by the security agents of [3]. We propose that the higher-level security issues considered in this paper will be dealt with by the *IMPACT* agents themselves, and not delegated to separate servers.

Security agents are also used in Distributed Object Kernel (DOK) project [55] for enforcing security policies in distributed and heterogeneous environments. There are three levels of agents. Top level agents are aware of all the activities that are happening in the system (or have already happened). Based on this information the agents of the top layer delegate functions to the appropriate agents. In the environments which we consider, agents cannot have information on all the activities that are happening and each agent should maintain its data and action security.

He et al. [26] proposed to implement the authorities of authentication verification service systems as autonomous software agents, called security agents, instead of building a static monolithic hierarchy as in the traditional Public Key Infrastructure (PKI) implementations. One of the open questions they present is: "How to define a suitable language for users to describe their security policy and security protocols so that the agent delegates of a user can safely transact electronic business on his behalf?" We believe that the language and framework presented in this paper can be used for such purpose, in addition to the original purpose of programming individual agents to maintain their data and action security.

Foner [23] discusses security problems in a multi-agent matchmaker system named Yenta.

IMPACT agents do not have access to other agents' data as Yenta's agents have. Each agent is responsible for its own data security. We believe that this approach will lead to more secure multi-agents systems.

Soueina et al. [50] present a language for programming agents acting in multi-agent environments. It is possible to give an agent commands such as "lie(action())" indicating that lying may be needed when the action is performed, "zone(action*)" that can be used to classify some agents as being hostile etc. Their work is based on first order logic and on concepts from game theory, but no formal semantics is given.

Other distributed object-oriented systems provide some security services. *CORBA* [45], an object request broker framework, provides security services, such as identification and authentication of human users and objects, and security of communication between objects. These services are not currently provided by *IMPACT*, and their implementation is left for future work. *CORBA* provides some simple authorization and access control. Our model allows the application of more sophisticated security policies using the ideas of approximations of agents' beliefs, state and consequence operations.

Zeng and Wang [66] proposed an Internet conceptual security model using Telos. They try to detect attacks based on monitoring and analyzing of audit information. In their framework a designer can construct an ontology of Internet security and then develops a set of rules for security maintenance. Their examples consider identifying security problems by analyzing network transactions. It is not clear from the papers how their rules will be used to preserve security and they do not consider data security problems.

Concordia is a framework for development and management of network-efficient mobile agent applications for accessing information anytime, anywhere and on any device supporting Java. Agent protection in *Concordia* [33] refers to the process of protecting agent's contents during transmission over the net. Prior to transmission an agent's byte-codes, member data and state information are encrypted through a combination of symmetric and public cryptography. In order to provide reliability, *Concordia* employs a persistent store to periodically checkpoint an agent. But, this on-disk representation may impose security risks, hence *Concordia* also encrypts this on-disk representation of an agent.

Sloman, Lupu and their colleagues [38, 39, 64] developed a role-based security model for distributed object systems in a large-scale, multi-organizational enterprise. In their model a role can be defined in terms of the authorization and obligation policies. Such policies specify what actions an agent or a person having this role is permitted or is obliged to do on a set of target objects. This permits individuals to be assigned or removed from positions without respecifying policies for the role. In particular, the authorization policies are used for access control and the obligation policies define actions to be performed by administrators or security components when events such as security violations are detected, e.g., the security administrator must investigate all sequences of 5 login failures from the same source.

We also presented a language which enables a designer to specify the actions the agent is obliged, forbidden, or allowed to take. In addition, we presented a theory and mechanisms in which an agent's state, history and beliefs dynamically effect the data it can access and the services available to it. On the other hand, we do not support role assignments and therefore "policies" should be specified for each individual agent.

11 Conclusions

As more and more "agent" applications are being built and deployed on the Internet, and as many multiagent applications involve "teams" of cooperating agents that dynamically form coalitions, there is a growing realization that security could be a problem.

In this paper, we have taken a set of first steps towards addressing how an agent developer can encode security mechanisms into an agent. Specifically, we have made the following contributions:

1. We have presented a (very) abstract definition of an agent and shown that for such agents to maintain security, several types of mathematical structures (history, consequence operation, etc.) need to be maintained.
2. As these structures often require an agent $\mathfrak{a}$ to have information about arbitrary agents $\mathfrak{b}$, and as this information may be hard to obtain in most practical applications, we have developed the concept of an "approximation" of this information, which leads to a notion of "approximate" security. We show that approximate security leads to security (under appropriate conditions).
3. We then provide a number of undecidability results showing that the general problem of maintaining data/action security is undecidable.
4. Then we propose a rule based logical language within which an agent developer may express approximations that his agent will use to approximate other agents.
5. We present algorithms for static and dynamic security checking which may be used once the agent developer has specified the approximation he or she wishes to use. We show that these algorithms are sound and complete and that (under appropriate assumptions) they have polynomial data complexity.

We conclude with a description of some future work.

- Agents must maintain data security in the presence of incomplete information about what other agents know. This explains the need for all the "approximation" languages in our framework. The agent developer is responsible for writing specifications approximating other agents. Methodologies to do this need to be developed.
- The current framework already provides some mechanisms by which an agent may protect itself from new agents. For example, an agent $\mathfrak{a}$ may maintain rules saying that for all unknown' agents $\mathfrak{b}$, certain rules apply. However, methods are needed to formally incorporate "new" agents into a multiagent system and to assess the impact of such new agents on security.
- A third problem is that of trading off security vs. use of system resources such as bandwidth and load. The more security requirements an agent $\mathfrak{a}$ is to enforce, the less resources (time, load) it can devote to the actual services it provides.

We hope the work presented in this paper lays a theoretical underpinning for investigating these questions in the future.

Acknowledgments

This work was supported by the Army Research Laboratory under contract number DAAL01-97-K0135, by the Army Research Office under contract DAAD190010484, by DARPA/RL contract number F306029910552, by an NSF Young Investigator award IRI-93-57756, and by NSF grants IIS-9820657 and IIS-9907482.

References

[1] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. Knoblock. Retrieving and Integrating Data From Multiple Information Sources. *International Journal of Intelligent Cooperative Information Systems*, 2(2):127–158, 1993.

[2] K. Arisha, F. Ozcan, R. Ross, V. S. Subrahmanian, T. Eiter, and S. Kraus. IMPACT: A Platform for Collaborating Agents. *IEEE Intelligent Systems*, 14:64–72, March/April 1999.

[3] F. Bergadano, A. Puliafito, S. Riccobene, and G. Ruffo. Java-based and secure learning agents for information retrieval in distributed systems. *INFORMATION SCIENCES*, 113(1-2):55–84, January 1999.

[4] S. Berkovits, J. Guttman, and V. Swarup. Authentication for Mobile Agents. In G. Vigna, editor, *Mobile agents and security*, volume 1419 of *Lecture Notes in Computer Science*, pages 114–136. Springer-Verlag, New York, NY, 1998.

[5] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. A Temporal Access Control Mechanism for Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):67–80, 1996.

[6] E. Bertino, P. Samarati, and S. Jajodia. Authorizations in relational database management systems. In *Proceedings of the 1st ACM Conference on Computer and Communication Security*, Fairfax, VA, November 1993.

[7] P. Bieber and F. Cuppens. A definition of secure dependencies using the logic of security. In *Proc. of the Computer Security Foundations Workshop IV*. IEEE Computer Society Press, 1991.

[8] E. J. Bina, R. M. McCool, V. E. Jones, and M. Winslett. Secure Access to Data over the Internet. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems (PDIS 94)*, pages 99–102, Austin, Texas, 1994. IEEE-CS Press.

[9] P. Bonatti, S. Kraus, J. Salinas, and V. S. Subrahmanian. Data Security in Heterogenous Agent Systems. In M. Klusch, editor, *Cooperative Information Agents*, pages 290–305. Springer-Verlag, 1998.

[10] P. Bonatti, S. Kraus, and V. S. Subrahmanian. Foundations of Secure Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):406–422, June 1995.

[11] R. Campbell and T. Qian. Dynamic Agent-based Security Architecture for Mobile Computers. In *The Second International Conference on Parallel and Distributed Computing and Networks (PDCN'98)*, Australia, December 1998.

[12] S. Castano, M. G. Fugini, G. Martella, and P. Samarati. *Database Security.* Addison Wesley, 1995.

[13] Cattell, R. G. G., editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, 1994.

[14] B.F. Chellas. *Modal Logic.* Cambridge University Press, 1980.

[15] D. M. Chess. Security in Agents Systems, 1996. `http://www.av.ibm.com/InsideTheLab/Bookshelf/ScientificPapers/`.

[16] D. M. Chess. Security Issues in Mobile Code Systems. In G. Vigna, editor, *Mobile agents and security*, volume 1419 of *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, New York, NY, 1998.

[17] M. Crosbie and E. Spafford. Applying genetic programming to intrusion detection. In *Proceedings of the AAAI 1995 Fall Symposium series*, November 1995.

[18] F. Cuppens. A modal logic framework to solve aggregation problems. In S. Jajodia and C. Landwehr, editors, *Database Security, 5: Status and Prospects.* North Holland, 1992.

[19] T. Eiter and V. S. Subrahmanian. Heterogeneous Active Agents, II: Algorithms and Complexity. *Artificial Intelligence*, 108(1-2):257–307, 1999.

[20] Thomas Eiter, V. S. Subrahmanian, and Georg Pick. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence*, 108(1-2):179–255, 1999.

[21] O. Etzioni and D. Weld. A Softbot-Based Interface to the Internet. *Communications of the ACM*, 37(7):72–76, 1994.

[22] W. M. Farmer, J. D. Guttag, and V. Swarup. Security for Mobile Agents: Authentification and State Appraisal. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, *Proceedings of the Fourth ESORICS*, volume 1146 of *Lecture Notes in Computer Science*, pages 118–130. Springer-Verlag, Rome, Italy, September 1996.

[23] L. N. Foner. A Security Architecture for Multi-Agent Matchmaking. In *Second International Conference on Multi-Agent Systems (ICMAS96)*, Japan, 1996.

[24] S. Fritzinger and M. Mueller. Java Security, 1996. `http://java.sun.com/docs/white/index.html`.

[25] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in Multiple-language, Mobile-Agent System. In G. Vigna, editor, *Mobile agents and security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, New York, NY, 1998.

[26] Q. He, K. P. Sycara, and T. W. Finin. Personal Security Agent: KQML-Based PKI. In K. P. Sycara and M. Wooldridge, editors, *Proceedings of the 2nd International Conference on Autonomous Agents (AGENTS-98)*, pages 377–384, New York, May 1998. ACM Press.

[27] N. Heintze and JD. Tygar. A model for secure protocols and their compositions. *IEEE Transactions on Software Engineering*, 22(1):16–30, January 1996.

[28] F. Hohl. An Approach to Solve the Problem of Malicious Hosts in Mobile Agent Systems. `http://inf.informatik.uni-stuttgart.de:80/ipvr/vs/mitarbeiter/hohlfz.en%gl.html`, 1997.

[29] M. Hughes. Application and enterprise security with the JAVA^{TM} 2 platform, 1998. `http://java.sun.com/events/jbe/98/features/security.html`.

[30] M. Huhns and M. Singh, editors. *Readings in Agents.* Morgan Kaufmann, 1997.

[31] S. Jajodia and R. Sandhu. Toward a Multilevel Relational Data Model. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Denver, Colorado, May 1991.

[32] N. R. Jennings. Controlling Cooperative Problem Solving in Industrial Multi-Agent Systems Using Joint Intentions. *Artificial Intelligence*, 75(2):1–46, 1995.

[33] R. Koblick. Concordia. *Communications of the ACM*, 42(3):96–97, March 1999.

[34] S. Kraus. Negotiation and Cooperation in Multi-Agent Environments. *Artificial Intelligence, Special Issue on Economic Principles of Multi-Agent Systems*, 94(1-2):79–98, 1997.

[35] Y. Labrou and T. Finin. A Semantics Approach for KQML – A General Purpose Communications Language for Software Agents. In *Proceedings of the International Conference on Information and Knowledge Management*, pages 447–455, 1994.

[36] Y. Labrou and T. Finin. Semantics for an Agent Communication Language. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 199–203, Providence, RI, 1997.

[37] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Germany, 1984, 1987.

[38] E. Lupu and M. Sloman. Towards a Role-based Framework for Distributed Systems Management. *Journal of Network and Systems Management*, 5(1):5–30, 1997.

[39] E. C. Lupu and M. S. Sloman. Reconciling Role Based Management and Role Based Access Control. In *Second Role Based Access Control Workshop (RBAC'97)*, pages 135–141, George Mason University, Virginia, 1997.

[40] P. Maes. Agents that Reduce Work and Information Overload. *Communications of the ACM*, 37(7):31–40, 1994.

[41] J. Millen and T. Lunt. Security for Object-Oriented Database Systems. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1992.

[42] B. Moulin and B. Chaib-Draa. An Overview of Distributed Artificial Intelligence. In G. M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 3–55. John Wiley & Sons, 1996.

[43] G. C. Necula and P. Lee. Research on Proof-Carrying Code on Mobile-Code Security. In *Proceedings of the Workshop on Foundations of Mobile Code Security*, 1997. `http://www.cs.cmu.edu/~necula/pcc.html`.

[44] T. Nishigaya. Design of Multi-Agent Programming Libraries for Java. `http://www.fujitsu.co.jp/hypertext/free/kafka/paper`, 1997.

[45] OMG. CORBAServices: Common Services Specification. Technical Report 98-12-09, OMG, December 1998. `http://www.omg.org/`.

[46] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter: Designing Conventions for Automated Negotiation Among Computers.* MIT Press, Boston, 1994.

[47] T. Sander and C. Tschudin. Protecting Mobile Agents Against Malicious Hosts. In G. Vigna, editor, *Mobile agents and security*, volume 1419 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, New York, NY, 1998.

[48] H. J. Schumacher and S. Ghosh. A fundamental framework for network security. *Journal of Network and Computer Applications*, 20(3):305–322, July 1997.

[49] J. Siegal. *CORBA Fundementals and Programming.* John Wiley & Sons, New York, 1996.

[50] S. O. Soueina, B. H. Far, T. Katsube, and Z. Koono. MALL: A multi-agent learning language for competitive and uncertain environments. *IEICE TRANSACTIONS ON INFORMATION AND SYSTEMS*, 12:1339–1349, 1998.

[51] W. Stallings. *Title Network and Internetwork Security: Principles and Practice.* Prentice-Hall, Englewood Cliffs, 1995.

[52] V. S. Subrahmanian. Amalgamating Knowledge Bases. *ACM Transactions on Database Systems*, 19(2):291–331, 1994.

[53] V.S. Subrahmanian, P. Bonatti, J. Dix, T. Eiter, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems: Theory and Implementation.* MIT Press, 2000. (To appear).

[54] H. Tai and K. Kosaka. The Aglets Project. *Communications of the ACM*, 42(3):100–101, March 1999.

[55] Z. Tari. Using agents for secure access to data in the Internet. *IEEE Communications Magazine*, 35(6):136–140, June 1997.

[56] A. Tarski. *Logic, Semantics, Metamathematics.* Hackett Pub Co, January 1981.

[57] C. Thirunavukkarasu, T. Finin, and J. Mayfield. Secret Agents – A Security Architecture for the KQML Agent Communication Language. In *Intelligent Information Agents Workshop, held in conjunction with Fourth International Conference on Information and Knowledge Management CIKM'95*, Baltimore, MD, November 1995.

[58] G. Vigna. Cryptographic Traces for Mobile Agents. In G. Vigna, editor, *Mobile agents and security*, volume 1419 of *Lecture Notes in Computer Science*, pages 137–153. Springer-Verlag, New York, NY, 1998.

[59] G. Vigna, editor. *Mobile agents and security.* Springer-Verlag, New York, NY, 1998. Lecture Notes in Computer Science, Volume 1419.

[60] M. Wellman. A Market-Oriented Programming Environment and its Application to Distributed Multicommodity Flow Problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[61] G. Wiederhold. Intelligent Integration of Information. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 434–437, Washington, DC, 1993.

[62] M. Winslett, K. Smith, and X. Qian. Formal Query Languages for Secure Relational Databases. *ACM Transactions on Database Systems*, 19(4):626–662, December 1994.

[63] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Reviews*, 10(2), 1995.

[64] N. Yialelis, E. Lupu, and M. Sloman. Role-Based Security for Distributed Object Systems. In *IEEE WET-ICE*, Stanford, 1996.

[65] M. Zapf, H. Mueller, and K. Geihs. Security requirements for mobile agents in electronic markets. *Lecture Notes in Computer Science*, 1402:205–217, 1998.

[66] L. Zeng and H. Wang. Towards a Multi-Agent Security System: A Conceptual Model for Internet Security. In *Proceedings of Fourth AIS (Association for Information Systems) Conference*, Baltimore, Maryland, August 1998.

A Appendix: Feasible, Rational, and Reasonable Status Sets

This appendix provides in succinct form the definition of various concepts of status sets from [20], to which the reader is referred for more information.

Definition A.1 (Status Set) A *status set* is any set S of ground action status atoms over the values from the type domains of a software package $\mathcal{S}$. For any operator $Op \in \{\mathbf{P}, \mathbf{Do}, \mathbf{F}, \mathbf{O}, \mathbf{W}\}$, we denote by $Op(S)$ the set $Op(S) = \{\alpha \mid Op(\alpha) \in S\}$.

Definition A.2 (Operator $\mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S)$) *Let $\mathcal{P}$ be an agent program and $\mathcal{O}$ be an agent state. Then, $\mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S) = \{Head(r\theta) \mid r \in \mathcal{P},\ R(r,\theta,S) \text{ is true on } \mathcal{O}\}$, where the predicate $R(r,\theta,S)$ is true iff (1) $r\theta : A \leftarrow \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n$ is a ground rule, (2) $\mathcal{O} \models \chi$, (3) if $L_i = Op(\alpha)$ then $Op(\alpha) \in S$, and (4) if $L_i = \neg Op(\alpha)$ then $Op(\alpha) \notin S$, for all $i \in \{1,\dots,n\}$.*

Definition A.3 ($\mathbf{A\text{-}Cl}(S)$) *A status set S is* deontic and action closed, *if for every ground action α, it is the case that $(DC1)$ $\mathbf{O}\alpha \in S$ implies $\mathbf{P}\alpha \in S$, $(AC1)$ $\mathbf{O}\alpha \in S$ implies $\mathbf{Do}\,\alpha \in S$, and $(AC2)$ $\mathbf{Do}\,\alpha \in S$ implies $\mathbf{P}\alpha \in S$.*

For any status set S, we denote by $\mathbf{A\text{-}Cl}(S)$ the smallest set $S' \supseteq S$ such that S' is closed under $(AC1)$ and $(AC2)$, i.e., action closed.

Definition A.4 (Feasible Status Set) Let $\mathcal{P}$ be an agent program and let $\mathcal{O}$ be an agent state. Then, a status set S is a *feasible status set* for $\mathcal{P}$ on $\mathcal{O}$, if $(S1)$-$(S4)$ hold:

$(S1)$ $\mathbf{App}_{\mathcal{P},\mathcal{O}_S}(S) \subseteq S$;

$(S2)$ For any ground action α, the following holds: $\mathbf{O}\alpha \in S$ implies $\mathbf{W}\alpha \notin S$, and $\mathbf{P}\alpha \in S$ implies $\mathbf{F}\alpha \notin S$.

$(S3)$ $S = \mathbf{A\text{-}Cl}(S)$, i.e., S is action closed;

$(S4)$ The state $\mathcal{O}' = \mathbf{conc}(\mathbf{Do}\,(S), \mathcal{O})$ which results from $\mathcal{O}$ after executing (according to some execution strategy $\mathbf{conc}$) the actions in $\mathbf{Do}\,(S)$ satisfies the integrity constraints, i.e., $\mathcal{O}' \models \mathcal{IC}$.

Definition A.5 (Groundedness; Rational Status Set) A status set S is *grounded*, if no status set $S' \neq S$ exists such that $S' \subseteq S$ and S' satisfies conditions $(S1)$–$(S3)$ of a feasible status set. A status set S is a *rational status set*, if S is a feasible status set and S is grounded.

Definition A.6 (Reasonable Status Set) Let $\mathcal{P}$ be an agent program, let $\mathcal{O}$ be an agent state, and let S be a status set.

1. If $\mathcal{P}$ is positive, i.e., no negated action status atoms occur in it, then S is a *reasonable status set* for $\mathcal{P}$ on $\mathcal{O}$, iff S is a rational status set for $\mathcal{P}$ on $\mathcal{O}$.
2. The reduct of $\mathcal{P}$ w.r.t. S and $\mathcal{O}$, denoted by $red^S(\mathcal{P},\mathcal{O})$, is the program which is obtained from the ground instances of the rules in $\mathcal{P}$ over $\mathcal{O}$ as follows.

(a) Remove every rule r such that $Op(\alpha) \in S$ for some $\neg Op(\alpha)$ in the body of r;

(b) remove all negative literals $\neg Op(\alpha)$ from the remaining rules.

Then S is a *reasonable status set* for $\mathcal{P}$ w.r.t. $\mathcal{O}$, if it is a reasonable status set of the program $red^S(\mathcal{P}, \mathcal{O})$ with respect to $\mathcal{O}$.