Annotated Sequence Diagrams

Jonathan K. Millen John D. Ramsdell

February 5, 2010

1 Introduction

An *annotated sequence diagram (ASD)* is a paradigm for a distributed logical computation in a system of locally communicating components. It can support conclusions about properties normally satisfied by the cooperative activity of different components, and about the consequences in case individual components have been corrupted.

We assume that communication between components is reliable and authenticated. This is the case, for example, if components are virtual machines and communication between them is controlled by a hypervisor or virtual machine monitor. An ASD defines a fixed sequence of actions for each component, which include internal computations and message exchanges with other components. The effects of these actions are expressed using annotations stating the properties that a component may assume about other components and its own data.

Our motivating application is to the boot-up procedure of a computer platform equipped with a Trusted Platform Module (TPM). The launch sequence is a succession of specially designed software modules that load, measure, and execute the subsequent modules in a way that is supposed to establish a "chain of trust". A fully adequate background on TPMs and supporting system architecture is beyond the scope of this paper, but we can give a small example of the kind of problem we address.

One of the features of a TPM is some "non-volatile storage" (NVR) in which the manufacturer or system builder can write some data authorized by a responsible party referred to as the system "owner". The NVR may contain, among other things, the hash of a public key used to authenticate other data approved by the owner. We will show later, in Section 3.2, how an ASD can express several aspects and applications of this mechanism:

- retrieving the approved hash from the NVR

- computing and comparing the hash of some code to be authenticated
- inferring trust in the component whose code was just authenticated
- finding the consequences if the component that performed the hash is corrupted

and other related issues.

In the remainder of this section we give an overview of how ASDs are presented, both visually and syntactically. In Section 2 we give a semantics for ASDs using multiset rewriting (MSR). Our approach is to specify, using deduction rules, how an ASD is compiled into a set of multiset rewrite rules. The resulting multiset rewriting system models the way the system may behave. We have written a Prolog program that performs the compilation and interprets the MSR rules. Starting with a given initial state of the multiset, it determines how the multiset may evolve, and consequently which annotations will be satisfied for a given ASD that may have some corrupted components. This way, we can perform some rudimentary model checking to establish claimed properties of a system design or catch some logical errors.

Section 3 gives some examples, both an abstract example illustrating some fine points of ASD methodology, and a more realistic, though simplified, example illustrating the TPM chain of trust application mentioned above.

In Section 4 we discuss related work in applications of message sequence diagrams and security protocol analysis, together with a summary of the main points of the paper.

1.1 ASD Overview

An ASD has a sequence of *steps*, each of which is either a *computation* $\mathsf{c}(p, \phi)$, an *inference* $\mathsf{i}(p, \phi)$, or a *message* $\mathsf{t}(p, q, \vec{x}, \phi)$ from p to q with data $\vec{x}$ implying an assertion ϕ. Components are represented by *principals* p and q, etc. Principals should be thought of as unique entities rather than roles, since corruption may be introduced independently to different components running different instances of the same role. Inferences are based on application-specific *inference rules* expressed in Prolog-like Horn clause form.

An ASD can be pictured as in Figure 1, where each principal labels a vertical line called a *timeline*, and annotations and messages are indicated in sequence from the top down, associated with the principal or principals involved. The step-sequence specification excludes some asynchronous be-

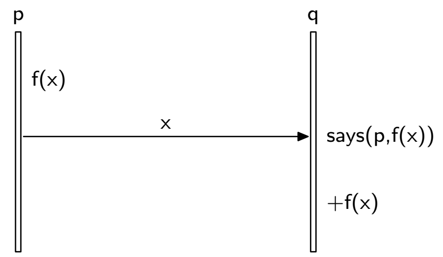

Figure 1: ASD for $\langle \mathsf{c}(p, f(x)), \mathsf{t}(p, q, \langle x\rangle, f(x)), \mathsf{i}(q, f(x))\rangle$

havior, such as that illustrated in Figure 2. However, the global ordering of steps is only partial, because our semantics allows for some concurrency across different components.

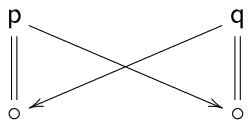

Figure 2: Non-ASD Behavior

1.2 Definitions

An ASD is a pair (S, I) where S is a sequence of steps and I is a set of inference rules. Recall that there are three kinds of steps: computations, inferences, and messages. A computation step is a term $\mathsf{c}(p, \phi)$ where p is a *principal* and ϕ is an *atomic formula.* A principal is a name for a component.

An atomic formula has the form $f(\vec{x})$, where f is a predicate symbol from an application-specific set, and $\vec{x} = \langle x_1, ..., x_n\rangle$ is the sequence of arguments. The arguments x_i represent data objects. Each argument is a *system variable* or a data constant.

An atomic formula is one kind of *predicate.* The other kind is an *assertion.* An assertion is a formula $\mathsf{s}(p, \phi)$ (read p says ϕ) where ϕ is a predicate.

An inference is a step $\mathsf{i}(p, \phi)$ (read p infers ϕ) where ϕ is a predicate. The *inference rules* I are specified in a Horn clause form $\vec{\psi} \rightarrow \phi$.

A message step from p to q is an expression $\mathsf{t}(p, q, \vec{x}, \phi)$ where p and q are principals such that $p \neq q$. In this step, $\vec{x}$ constitutes the data or content of the message, and ϕ is a predicate. (Here, t stands for "transmit"; we will use m later for the message fact in the multiset.) The predicate may be omitted and the message data may be the empty sequence. The predicate given in the message step is not actually sent from one principal to the other; it supports an inference by the receiver that the sender asserts that predicate, expressed as an assertion: $\mathsf{s}(p, \phi)$. The inference is based on the known step sequence of the ASD—it is legal only if there is a previous step $\mathsf{c}(p, \phi)$ or $\mathsf{i}(p, \phi)$.

When two principals share the use of a system variable, it is usually an indication that the system designer intends that variable to be bound to the same value in both components. This similar to the convention used in "Alice-Bob" specifications of authentication protocols. In the ASD semantics, agreement is tested in message steps, but may fail as a result of the corruption of some components.

1.3 Notation

In what follows, a finite sequence $\vec{x} = \langle x_1, \ldots, x_n \rangle$ is a function on $\{1, ..., n\}$ with $x_i = \vec{x}(i)$. Its length is $|\vec{x}| = n$ and its range is $[\vec{x}] = \{x_i \mid i \leq |\vec{x}|\}$. We will write $f(x_1, ..., x_n)$ rather than $f(\langle x_1, ..., x_n \rangle)$. We may write f_a for $f(a)$ for any single-argument function f. $\vec{x}^\frown\vec{y}$ is the concatenation of sequences $\vec{x}$ and $\vec{y}$, and $a^\frown\vec{x}$ means $\langle a \rangle^\frown\vec{x}$ if a is not a sequence.

$\vec{x} \cup \vec{y}$ is a kind of vector union. It can be defined as a maximal subsequence of the concatenation $\vec{x}^\frown\vec{y}$ having no repeats and no constants; order doesn't matter. It has the property that $[\vec{x} \cup \vec{y}] = ([\vec{x}] \cup [\vec{y}]) \setminus D$ where D is the set of data constants.

S^* is the set of finite sequences of elements of a set S, and $\wp(S)$ is the set of subsets of S.

It is convenient to use the notation $f[x \mapsto t]$ for a function update, where

$$f[x \mapsto t](y) = \begin{cases} t & \text{if } x = y, \\ f(y) & \text{otherwise.} \end{cases}$$

2 MSR Generation

A multiset rewriting system is defined by a set of rewrite rules and a language for expressing "facts" in the multiset. The state of the system is the current multiset of facts. A multiset is an unordered set in which a fact may appear with some multiplicity greater than one. Rewrite rules are easy to implement for multisets.

2.1 MSR Syntax

Figure 3 presents the signature used in expressing MSR facts. In this model, a fact is either

- an atomic formula $\mathsf{a}(f, \vec{x})$,
- a belief $\mathsf{b}(p, \phi)$,
- the local state of a principal $\mathsf{h}(p, n, \vec{x})$, or
- a message $\mathsf{m}(p, q, \vec{x})$.

Note that MSR facts expressing atomic formulas are written using an "apply" operation $\mathsf{a}(f, \langle x_1, ..., x_n\rangle)$ rather than $f(x_1, ..., x_n)$, so that we can make a fixed finite list of the operation symbols at the top level of facts. However, when we present the formalism below, we will revert to writing $f(\vec{x})$ rather than $\mathsf{a}(f, \vec{x})$ for the sake of readability.

In order to express the particular atomic formulas that occur in ASD steps and inference rules, we need to allow constant symbols for predicates, principals, and other data, as many as needed.

Facts are always ground terms. An atomic formula is present in the multiset if it is a global assumption, available to all principals as a computational resource. A belief indicates that a particular principal has taken a ground formula into its local theory, and may use that fact in subsequent inferences. The local state of a principal includes the number of steps it has traversed (called the *height*) and the list of data values it has acquired for system variables. The association between values and system variables is implicit in the order in which they appear.

There are two particular predicate symbols that show up in our rewrite rules: g and n. These are called *integrity facts*. If $\mathsf{g}(p)$ is assumed, by placing $\mathsf{a}(\mathsf{g}, \langle p\rangle)$ in the initial multiset, the principal p is honest and follows its role in the ASD. If $\mathsf{n}(p)$ is assumed, the principal p is dishonest. A dishonest principal represents a corrupted or compromised component. It does not check beliefs or inferences, and when it sends a message, the data in the message may be chosen arbitrarily. A dishonest principal updates its local state, however, so that it will be able to send expected messages on cue. Exactly one of $\mathsf{g}(p)$ and $\mathsf{n}(p)$ must always occur in the multiset for each principal p.

Sorts: nat, psym, prin, data, pred, fact. Sort nat denotes the natural numbers and psym denotes a set of predicate symbols.
Subsorts: prin < data, atmf < pred, atmf < fact.
Operations:

$\mathsf{a}\colon \mathsf{psym} \times \mathsf{data}^* \to \mathsf{atmf}$	Atomic formula
$\mathsf{b}\colon \mathsf{prin} \times \mathsf{pred} \to \mathsf{fact}$	Belief
$\mathsf{s}\colon \mathsf{prin} \times \mathsf{pred} \to \mathsf{pred}$	Assertion
$\mathsf{h}\colon \mathsf{prin} \times \mathsf{nat} \times \mathsf{data}^* \to \mathsf{fact}$	Local state
$\mathsf{m}\colon \mathsf{prin} \times \mathsf{prin} \times \mathsf{data}^* \to \mathsf{fact}$	Message
(constants omitted)	

Figure 3: ASD Multiset Rewriting Signature

2.2 The Compilation Procedure

The compilation procedure is expressed as a set of deduction rules. The input to the ASD compilation procedure is a sequence S of steps and the set I of inference rules. The translation is expressed by the deduction system as a judgement of the form

$$\Delta, H, I, S \twoheadrightarrow R \tag{1}$$

where $H : \mathsf{prin} \to \mathsf{nat}$ is the local state number (or height) map and $\Delta : \mathsf{prin} \to \wp(X)$ is the local state context map, indicating which system variables are defined for principal p. The output of the compilation is the set R of rewrite rules generated by the step sequence S.

In the initial state H, each principal has zero height. That is, for all p, $H_p = 0$. The initial context Δ_p is normally empty, though the formalism does not exclude initial states in which some principals have some system variables already defined.

The first deduction rule says that MSR rules are generated from the steps sequentially, one step at a time, though a single step may generate more than one rule. The deduction for an individual step also updates the local state and context maps. We will see that the progression of local contexts in the compilation sequence is reflected, in different notation, in the multiset rewrite rules that are generated. The rewrite rules allow for concurrent execution in a different order, however, when they are exercised.

$$\frac{\Delta, H, I, s \twoheadrightarrow \Delta', H', \rho \quad \Delta', H', I, S \twoheadrightarrow R}{\Delta, H, I, s^\frown S \twoheadrightarrow \rho \cup R} \tag{2}$$

A rewrite rule is presented as $(F \longrightarrow G)$ where F and G are comma-separated lists of facts, with dummy pattern variables in place of some or all data constants. We will also have use for a quantified rule $(F \longrightarrow (\exists \vec{x})G)$. The semantics of rewrite rules is discussed in Section 2.6.

2.3 Inference Step

An inference step applies an inference rule by adding its conclusion as a belief when its hypotheses are present in the multiset as beliefs. If the principal is compromised, it does not check the predicate or update its beliefs.

$$\frac{(\vec{\psi} \rightarrow \phi) \in I}{\Delta, H, I, \mathsf{i}(p, \phi\sigma) \twoheadrightarrow \Delta, H', I, \rho} \tag{3}$$

where

$$\begin{aligned} H' &= H[p \mapsto H_p + 1] \\ \rho &= \{(\mathsf{h}(p, H_p, \vec{x}), \mathsf{b}(p, \vec{\psi}\sigma), \mathsf{g}(p) \longrightarrow \mathsf{h}(p, H'_p, \vec{x}), \mathsf{b}(p, \vec{\psi}\sigma), \mathsf{b}(p, \phi\sigma), \mathsf{g}(p)), \\ &\quad\ (\mathsf{h}(p, H_p, \vec{x}), \mathsf{n}(p) \longrightarrow \mathsf{h}(p, H'_p, \vec{x}), \mathsf{n}(p))\} \end{aligned}$$

and $\mathsf{b}(p, \vec{\psi})$ is an abbreviation for the set of facts $\mathsf{b}(p, \psi_1), ..., \mathsf{b}(p, \psi_k)$ if $k = |\vec{\psi}|$. The substitution σ instantiates the dummy pattern variables in the inference rule conclusion ϕ with system variables to get the step conclusion $\phi\sigma$.

There is an important observation to be made in applying rule (3): there may be more than one inference rule and substitution that produce the step conclusion $\phi\sigma$. If so, several judgements may arise from (3), and each one of them creates two rewrite rules that update H in the same way. The second (compromised-principal) one is always the same; so if there are k inference rules that apply to one inference step, there are $k + 1$ different rewrite rules produced for it.

2.4 Computation Step

In a computation step, a principal ensures that a predicate is true. Sometimes this causes system variables to become defined. Recall that the compilation context Δ_p is the set of system variables defined for p. Suppose that the step is $\mathsf{c}(p, \mathsf{hash}(x, y))$, and that $x \in \Delta_p$ but y is still undefined. The rewrite rule for this step causes p to locate a fact $\mathsf{hash}(x, y)$ and add a belief $\mathsf{b}(p, \mathsf{hash}(x, y))$. It also adds y to Δ_p. (Although a hash computation is a lookup in a pure rewrite system, an implementation would use an algorithm to compute it.)

A compromised principal, as in the inference step, will update its local state but not bother with recording a belief.

$$\frac{}{\Delta, H, I, \mathsf{c}(p, f(\vec{x})) \twoheadrightarrow \Delta', H', I, \rho} \tag{4}$$

where

$$\begin{array}{rcl}
\Delta' & = & \Delta[p \mapsto \Delta_p \cup [\vec{x}]] \\
H' & = & H[p \mapsto H_p + 1] \\
\rho & = & \{(\mathsf{h}(p, H_p, \vec{z}), f(\vec{x}), \mathsf{g}(p) \\
 & & \longrightarrow \mathsf{h}(p, H'_p, \vec{z} \cup \vec{x}), f(\vec{x}), \mathsf{b}(p, f(\vec{x})), \mathsf{g}(p)) \\
 & & (\mathsf{h}(p, H_p, \vec{z}), \mathsf{n}(p) \longrightarrow \mathsf{h}(p, H'_p, \vec{z} \cup \vec{x}), \mathsf{n}(p))\}
\end{array}$$

2.5 Message Step

The next deduction rule shows the effect of a message step. Four rewrite rules are generated. The first two represent the normal case for sender and receiver. The last two show what can happen if the sender or receiver is compromised. A compromised sender can put any values available to it into the message. A compromised receiver will, as usual, add the expected context but omit recording any beliefs.

If the message data contains a system variable x that already occurs in the receiver's context, the rule permits a transition only when the ground values agree.

$$\frac{[\vec{x}] \subseteq \Delta_p}{\Delta, H, I, \mathsf{t}(p, q, \vec{x}, \phi) \twoheadrightarrow \Delta', H', I, \rho} \tag{5}$$

where

$$\begin{array}{rcl}
\Delta' & = & \Delta[q \mapsto \Delta_q \cup [\vec{x}]] \\
H' & = & H[p \mapsto H_p + 1][q \mapsto H_q + 1] \\
\rho & = & \{(\mathsf{h}(p, H_p, \vec{z}), \mathsf{b}(p, \phi), \mathsf{g}(p) \\
 & & \longrightarrow \mathsf{h}(p, H'_p, \vec{z}), \mathsf{b}(p, \phi), \mathsf{m}(p, q, \vec{x}), \mathsf{g}(p)), \\
 & & (\mathsf{h}(q, H_q, \vec{z}), \mathsf{m}(p, q, \vec{x}), \mathsf{g}(q) \\
 & & \longrightarrow \mathsf{h}(q, H'_q, \vec{z} \cup \vec{x}), \mathsf{b}(q, \mathsf{s}(p, \phi)), \mathsf{g}(q)), \\
 & & (\mathsf{h}(p, H_p, \vec{z}), \mathsf{n}(p) \longrightarrow (\exists \vec{y})\, \mathsf{h}(p, H'_p, \vec{z}), \mathsf{n}(p), \mathsf{m}(p, q, \vec{y})), \\
 & & (\mathsf{h}(q, H_q, \vec{z}), \mathsf{n}(q), \mathsf{m}(p, q, \vec{x}) \longrightarrow \mathsf{h}(q, H'_q, \vec{z} \cup \vec{x}), \mathsf{n}(q))\}
\end{array}$$

In the third (compromised-p) rewrite rule, the existentially quantified system variables in $\vec{y}$ are chosen such that $[\vec{y}] \cap [\vec{z}] = \emptyset$, and $|\vec{y}| = |\vec{x}|$. One might wonder why the arbitrary value choices are made here, when the message is composed, rather than in computation steps, where variable values could be assigned freely. The reason is that this way, a dishonest principal can send different putative values for the same variable in messages to two other principals.

2.6 MSR Semantics

The initial multiset should have an initial local state $\mathsf{h}(p, 0, \langle\rangle)$ and an integrity predicate of either $\mathsf{g}(p)$ or $\mathsf{n}(p)$ for each principal p. It will also have some collection of atomic formulas $f(x)$ representing computational resources. Normally, even as the state progresses, it will be a set rather than a multiset, but it is possible for multiple copies of messages to appear.

Rewrite rules ($F \longrightarrow G$) with no quantifier are applied in the usual way. A rule is enabled if there is a substitution σ on pattern variables into data constants such that for each fact ϕ in F, $\phi\sigma$ occurs at least once in the multiset $\mathcal{S}$. Then the rule is applied by deleting one occurrence of each left-side fact $\phi\sigma$ from $\mathcal{S}$ and, for each ψ in G, adding one occurrence of $\psi\sigma$ to $\mathcal{S}$.

In a quantified rule ($F \longrightarrow (\exists \vec{x})G$), the substitution σ maps each x_i to some data constant, but these values are not restricted to fresh values, or "nonces". The universe of data constants that may instantiate existentially quantified system variables includes all those constants that occur in the initial multiset, as well as fresh data constants. This is a different interpretation of the existential quantifier from that used in [4].[1]

We expect that that a simulation or model-checking tool that exercises the rewrite rules will have some strategy for exploring the possibilities in a practical way. The tool or the user would have to look for ways to do data abstraction. Our Prolog implementation simply leaves the quantified variables as variables, and binds them opportunistically to enable later transitions.

One might ask whether the "arbitrary" values introduced by a compromised principal should have been restricted to values that can be computed from data already known to it. That is the approach used in security protocol modeling. But this model does not explicitly address management of secrets or cryptographic functions. Nevertheless, the power of a compromised component is limited, because it can only transmit data, not terms. Thus, if cert(p,kp,kc,s) means that s is a signature validated with a public key kc certifying that kp is the public key of p, that ground formula would be in the initial state, and could be checked by a relying party. However, ground instances of cert with other, incorrect combinations of data would (should) not be present. A component that receives a message associated with an assertion says(q, cert(p,kp',kc,s)) with the wrong public key would not accept the claim unless it trusted q.

2.7 Prolog System

We are experimenting with a Prolog program that accepts a Prolog-syntax variant of an ASD specification, displays the steps in a diagram like Figure 1, compiles the steps into rewrite rules, and exercises the rewrite rules from specified initial states. The program explores multiset state reachability.

An ASD specification has three parts: a list of initial states, a list of steps, and a list of inference rules. The input syntax is under development and is not stable, but for concreteness, Figure 4 shows an example. In Prolog syntax, it is natural to represent sequences by Prolog lists, with square brackets. Identifiers beginning with a lower-case letter are atoms, used for constants and predicate symbols. Identifiers beginning with an upper-case letter are Prolog variables, and are used in inference rules. Quoted upper-case symbols like `'X'` are system variables, which are treated as constants by Prolog. The quotes will later be stripped from the variables when the compiler puts them into rewrite rules—this is explained below.

The first two parts of the specification are combined in an `asd` term. The

[1]Should we have used $\forall$ instead of $\exists$?

```
asd([ [a(f,[x]),h(p,0,[]),a(g,[p]),h(p,0,[]),
                a(g,[q]),h(p,0,[]),a(g,[r])],
      [a(f,[x]),h(p,0,[]),a(n,[p]),h(p,0,[]),
                a(g,[q]),h(p,0,[]),a(g,[r])],
      [a(f,[x]),h(p,0,[]),a(n,[p]),h(p,0,[]),
                a(g,[q]),h(p,0,[]),a(n,[r])] ],
    [ c(p,a(f,['X'])),
      t(p,q,['X'],a(f,['X'])),
      t(q,r,[p],true),
      c(r,a(g,[p])),
      t(r,q,[p],a(g,[p])),
      i(q,a(g,[p])),
      i(q,a(f,['X']))]).

irs([ ir(b(P,a(g,[Q])),[b(P,s(r,a(g,[Q])))]),
      ir(b(P,A),[b(P,s(Q,A)),b(P,a(g,[Q]))]) ]).
```

Figure 4: Prolog Input, Trust Example

actual Prolog tool requires only the atomic facts to be listed, and inserts the standard initial heights and integrity facts automatically. Integrity facts can be changed (g to n and back) interactively. ASD steps are represented as in the text, with atomic facts in the full apply form. Each inference rule has the form ir(*conclusion*,[*hypotheses*]). When the same list of hypotheses justify two or more conclusions, the conclusions may appear together as a list in a single rule.

The steps in Figure 4 are displayed by the tool as in Figure 5.

Loading, compilation, and running are controlled by dialog window buttons. Compilation writes initial states and rewrite rules out to a file, and running the ASD generates one or more possible state sequences starting with each given initial state. The final state in each sequence is displayed by showing the beliefs of each principal on a diagram as though they were computation steps. The built-in Prolog backtracking mechanism is used to generate the possible state sequences.

Generation of state sequences is handled with a very simple rewrite system. The multiset is a Prolog list of facts. Rewriting is handled by a Prolog predicate that is an inessential variation of the one shown in Figure 6. Given a rewrite rule ($L \rightarrow R$), and a fact list S_1, the predicate tests if L can be instantiated so that its elements all occur in S_1, and if so, removes them and appends R (with the same instantiation) to get S_2.

The L, R pairs generated by the compilation are incorporated into a set

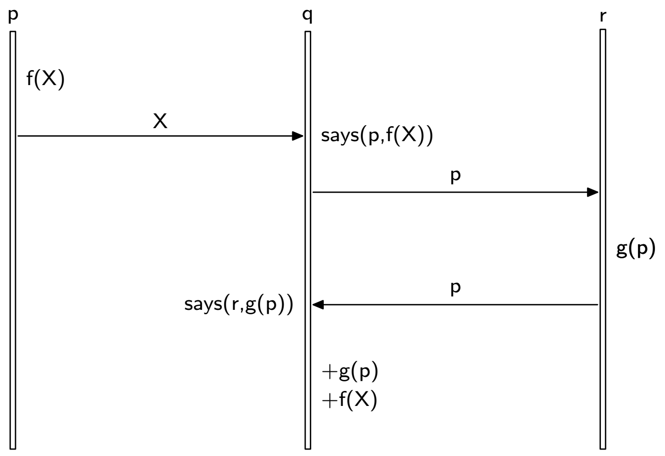

Figure 5: Display, Trust Example

```
rewrite(S1,S2,L,R) :-
  subset(L,S1),
  subtract(S1,L,S0),
  append(R,S0,S2).
```

Figure 6: Rewrite Predicate

of clauses defining the next-state transition predicate `rule`. For example, the first computation step for `p` would generate the following: In this clause,

```
rule(S1,S2) :- rewrite(S1,S2,
  [h(p,0,[]),a(f,[X]),a(g,[p])],
  [h(p,1,[X]),a(f,[X]),a(g,[p])]).
```

Figure 7: Rewrite Rule for a Computation Step

the system variable `X` no longer has quotes, and is now a Prolog variable. Thus, when `rewrite` is called as above, `X` is instantiated with whatever data constant occurs in the multiset as an argument of `f`; it has become a pattern variable for the rewrite rule.

3 Analysis Examples

This section has some further observations about the example in Figures 4 and 5 and then introduces a more realistic example arising from our TPM application.

3.1 Trust Example Analysis

First, note that, in order for any computation step to be successful in a run, the predicate must be instantiated in a ground term appearing in the initial state. This is why there is a ground atomic predicate fact for $f(x)$ in Figure 4. The analyst must supply symbolic ground facts to allow runs to be generated.

The story motivating the diagram is this: principal q wishes to receive a value X that principal p tells it satisfies $f(X)$. However, q first obtains assurance from principal r that principal p is not compromised. Component r is a measurement agent that is able to determine this. Principal q trusts r implicitly. Although the message from r to q is empty, the fact that there was a message from r is enough to convey the assurance from r about p. (The message step would have been rejected by the compiler if the check by r on $\mathsf{g}(p)$ had been omitted.)

Before going further, let us examine the inference rules that come into play. There are two, which may be paraphrased as follows:

- $\mathsf{b}(P, \mathsf{s}(r, \mathsf{g}(Q))) \to \mathsf{b}(P, \mathsf{g}(Q))$. That is, if P believes that r says that Q is good, then P believes that Q is good.
- $\mathsf{b}(P, \mathsf{s}(Q, A)), \mathsf{b}(P, \mathsf{g}(Q)) \to \mathsf{b}(P, A)$. That is, if P believes that Q says any predicate A, and P believes that Q is good, then P believes A.

The second rule is the *trust rule*, a very powerful rule, to the effect that any principal believes whatever any trusted principal says. This may seem dangerous, since trust is usually limited to certain areas of authority or competence, but the limitations can be applied in the steps rather than the rule. An inference step that uses this rule occurs in some particular place in the diagram, where the system designer is aware of its full instantiation, including the current state, the particular conclusion, and the principal being trusted for it. If this still seems objectionable, however, one may omit this rule from the input specification; it is not built in.

The first rule is almost a special case of the trust rule. The difference is that the trusted principal r is named with a constant, without presupposing any computational or other check on the goodness of r. Unlike the trust rule, this one can be applied when r is compromised. We will see the effect of this below.

When the example diagram is run, the Prolog ASD tool finds all of the complete state sequences resulting from each initial state it was given. A *trace* consists of the sequence of the beliefs of all good principals, in the order they were attained. The normal initial state produces the expected result, showing a display like Figure 8. Output displays do not have message arrows or distinguish inferences.

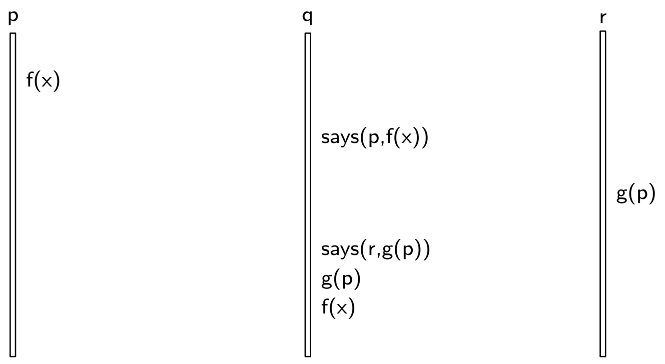

Figure 8: Output of Running ASD Example

When the initial state is changed so that p is compromised, another trace results in which p and r have no beliefs, but q has just one belief: that p says $f(X)$. Since p is compromised, it has no beliefs. And since r is good, it discovers p's compromise, and cannot attain the needed belief in p's goodness to proceed. In the trace, p's assertion to q is $f(X)$, where X is an uninstantiated variable, due to our Prolog tactic for handling the existential quantifier.

Another trace is found when p and r are both compromised. In this case, r sends the message to q implying that p is good. Because q accepts the word of r, q arrives at a belief in $f(X)$, with X still a variable in the Prolog output. The misplaced, unverified trust of q in r has enabled p to get q to believe property f of anything he likes. In this trace, p and q have no beliefs, but q has all its beliefs of q as shown in Figure 5.

3.2 TPM Application

We can now give a small taste of how sequence diagrams are used in the design analysis of a system that makes use of virtualization and a Trusted Platform Module (TPM). A high-level description of the architecture of this system is given in [3]. This is not the place to explain TPM details, but the reader can consult the Trusted Computing Group Website [9] for authoritative reference material. The textbook edited by Pearson [8] is still one of the best places to start to understand the basic concepts, though it does not cover more recent extensions and applications.

The two diagrams below show two connected fragments of a boot sequence in which several trusted domains (virtual machines) are loaded and launched in a prescribed order. One way in which the integrity of a domain is established is by an *authenticated build.* Such a domain is provisioned with a header containing a signed hash of the image. A trusted domain called the *domain builder* must first load and authenticate a public key used for the signatures. Then it can recompute and check the hash of another domain to be authenticated.

In the first diagram, showing the "get signing key" phase, the domain builder gets the hash of the authenticating key from the non-volatile storage (NVR) in the TPM, which is firmware initialized by an entity called the platform "owner". The key itself is stored in the domain builder's memory. The domain builder checks that this key is correct, leading to the inference $\mathsf{ks}(\mathsf{K})$ stating that K is the owner's signature key. The specification is in Figure 9 and the displayed diagram is in Figure 10. In these figures, the two principals are the domain builder db and the TPM tpm.

In Figure 9 there is only one initial state, and it does not include local states or integrity facts. The analysis tool supplies initial local states and integrity facts automatically.

When a specification is divided into fragments, as the TPM examples illustrate, there has to be some way to reflect assumptions that can be used by this fragment that are inherited from earlier fragments. One way to do this is to include a belief in the initial state, as is done in Figures 9 and 10. The inherited belief is that the TPM is honest. If this assumption had been included as an atomic fact, we would be unable to examine scenarios in which the TPM has been compromised but the domain builder still believes that it is honest.

The inference that the TPM is honest, shown as $+\mathsf{g}(\mathsf{tpm})$ in the display, would be invisible unless it was included explicitly as it is here. It is not a computation, because that would fail if the TPM were not actually honest.

```
asd([
       [b(db,a(g,[tpm])),
        a(load,[ks_hash_addr,a]),
        a(readNVR,[a,h]),
        a(load,[ks,k]),
        a(hash,[k,h])]
  ],
       [i(db,a(g,[tpm])),
        c(db,a(load,[ks_hash_addr,'A'])),
        t(db,tpm,[readNVR,'A'],true),
        c(tpm,a(readNVR,['A','H'])),
        t(tpm,db,['H'],a(readNVR,['A','H'])),
        i(db,a(readNVR,['A','H'])),
        c(db,a(load,[ks,'K'])),
        c(db,a(hash,['K','H'])),
        i(db,a(ks,['K']))
  ]).
irs([
        ir(b(P,A), [b(P,A)]),      % repeat rule
        ir(b(P,A),                 % trust rule
                [b(P,a(g,[Q])),
                b(P,s(Q,A))]),
        ir(b(db,a(ks,[K])),        % auth key retrieval
                [b(db,a(load,[ks_hash_addr,A])),
                b(db,a(readNVR,[A,H])),
                b(db,a(hash,[K,H]))])
   ]).
```

Figure 9: Get Signing Key: Specification

In order to succeed as an inference, we need a rule that allows an existing belief to be restated; that is the "repeat rule" seen in the specification.

If the diagram is run after selecting the TPM as compromised, the output shows all domain builder beliefs like this:

```
db
 ‖ -g(tpm)
 ‖ load(ks_hash_addr,A)
 ‖ readNVR(A,H)
 ‖ load(ks,K)
 ‖ hash(K,H)
 ‖ ks(K)
```

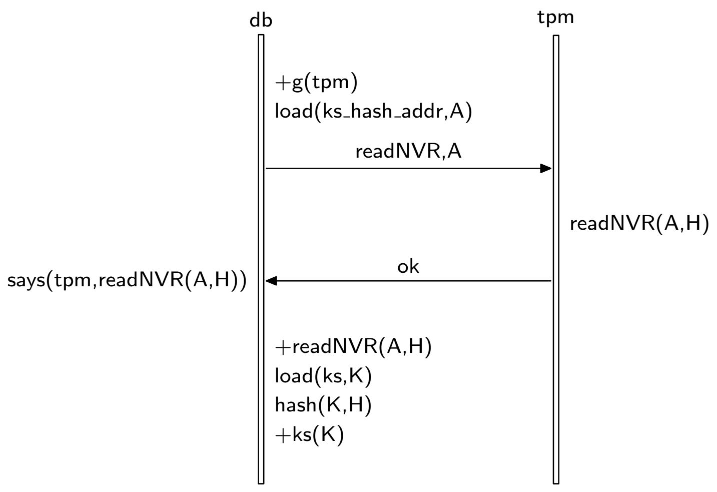

Figure 10: Get Signing Key: Display

This list indicates that the domain builder completed its normal state sequence, arriving at the final belief in K, but the negation -g(tpm) warns us that belief in the goodness of the TPM is inconsistent with the actual state facts. Any subsequent inferences that might have used this belief are therefore suspect. Because the system does not yet track which actual inferences are tainted by the false belief, we needed to state the inherited belief explicitly to call attention to its failure.

An authenticated build by the domain builder on request by another domain called the *control domain* is shown in the specification Figure 11 and the diagram Figure 12. The control domain names another component d that the domain builder loads and authenticates using the signature key. It reloads the signature key, but it carries over the belief that this key is correct by using a "ks carryover" rule stating that the loaded signing key is the correct one (satisfying the ks( ) predicate). This rule would not be necessary if we combined the two diagrams into a single larger one.

The domain builder loads the code C of d and uses the predicate sign to confirm that the signature produced by K on the actual code matches the signature S stored in the header and retrieved by auth. The control domain assumes via an initial belief that the domain builder is honest, so that it can infer integrity of the code of d. As before, if this belief is incorrect, running the ASD will mark the inference g(db) as false.

The ASD specification and analysis support addresses many other aspects of the design not illustrated in this brief sample, such as the way hashes and other measurements are stored in the TPM configuration registers and used later to support inferences about the system state when keys and other secrets are unsealed by TPM commands.

Much of the benefit of this kind of analysis comes not from the investigation of scenario variations after the specification is complete, but rather from the process of getting the specification working to begin with, with an adequate set of assumptions and inference rules. By making the assumptions and inference rules explicit, we enable the designers to focus on questions about why they are correct or what they have to do to make them correct. Inferences with the conclusion $\mathsf{g(p)}$, that a component is correct, honest, and uncompromised, are particularly worrisome– it difficult to defend such a conclusion with a clear conscience. In any case, the inference rules are up to the analyst.

4 Related Work

There have been many approaches to the formalization of message sequence diagrams. Some general semantic dimensions were suggested by Hausmann, *et al.* in [6]. With respect to some of the discriminators they propose, this model is of instances rather than roles, and implies a partial ordering of events. Time is not quantified. The ASD steps provide a specification rather than a scenario, in the sense that all legitimate event sequences are represented, although different initial states can be specified to investigate different scenarios.

Message sequence diagrams are supported by UML, leading to various approaches to formalize their use in UML, such as [1] and Section 8.1 of [7]. These incorporate an explicit object-oriented state structure for components that could probably be made to represent our annotations and state information. Our tradeoff is to give up full UML standard coverage but gain a more streamlined view of component trust and compromise.

Our model could also apply to a distributed system over a network, provided that cryptographic protocol mechanisms are used to implement authenticated channels. Indeed, most secure session establishment protocols have authentication as well as confidentiality as a goal.

Our MSR model is close to those used in the context of security protocol analysis, notably [4], but differs because our application can assume authenticated communication and VM separation, enabling us to focus on issues of inter-component trust, measurement, dependency, and compromise. The modal logic operators *believes* and *says* were previously employed in BAN logic [2], but the encryption-specific features of BAN have been removed to allow for application-specific predicates and inference rules. Our "Says" inferences resulting from message reception may be seen as a restricted instance of the Soundness condition in [5]. Our treatment of integrity facts appears to be new.

Our ongoing work is likely to emphasize improvements in the analysis software and more extensive worked application results. There is plenty of room for improvements in the Prolog tool to facilitate user interaction and automate aspects of property testing.

References

[1] D. Aredo. A framework for semantics of UML sequence diagrams in PVS. *J. Universal Computer Science* 8(7), 2002, pp. 674–697.

[2] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. on Computer Systems* 8(1), 1990, pp. 18–36.

[3] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. Principles of Remote Attestation. *Int. J. Information Security.* To Appear, 2010.

[4] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. *J. Computer Security* 12(2), 2004, 247–311.

[5] J. Guttman, F. J. Thayer, J. Carlson, J. Herzog, J. Ramsdell, and B. Sniffen. Trust management in strand spaces: a rely-guarantee method. *European Symposium On Programming*, 2004.

[6] J.Hausmann, J. Küster, and S. Sauer. Identifying Semantic Dimensions of (UML) Sequence Diagrams. *pUML* 2001, pp. 142–157.

[7] J. Jürgens. *Secure Systems Development with UML.* Springer, 2005.

[8] S. Pearson, ed. *Trusted Computing Platforms: TCPA Technology in Context.* Prentice Hall, 2002.

[9] Trusted Computing Group. `http://www.trustedcomputinggroup.org`.

```
asd([
      [ b(cd,a(g,[db])),
        a(load,[ks,k]),
        a(load,[d,c]),
        a(auth,[c,s]),
        a(sign,[c,k,s])]
      ],
      [ i(cd,a(g,[db])),
        t(cd,db,[d],true),
        c(db,a(load,[d,'C'])),
        c(db,a(auth,['C','S'])),
        c(db,a(load,[ks,'K'])),
        c(db,a(sign,['C','K','S'])),
        i(db,a(ks,['K'])),
        i(db,a(g,[d])),
        t(db,cd,[ok],a(g,[d])),
        i(cd,a(g,[d]))
      ]).

irs([
        ir(b(P,A), [b(P,A)]),             % repeat rule
        ir(b(P,A),                        % trust rule
                [b(P,a(g,[Q])),
                b(P,s(Q,A))]),
        ir(b(db,a(ks,[K])),               % ks carryover
                [b(db,a(load,[ks,K]))]),
        ir(b(db,a(g,[D])),          % domain authentication
                [b(db,a(load,[D,C])),
                 b(db,a(auth,[C,S])),
                b(db,a(ks,[K])),
                b(db,a(sign,[C,K,S]))
                ])
   ]).
```

Figure 11: Authenticated Build: Specification

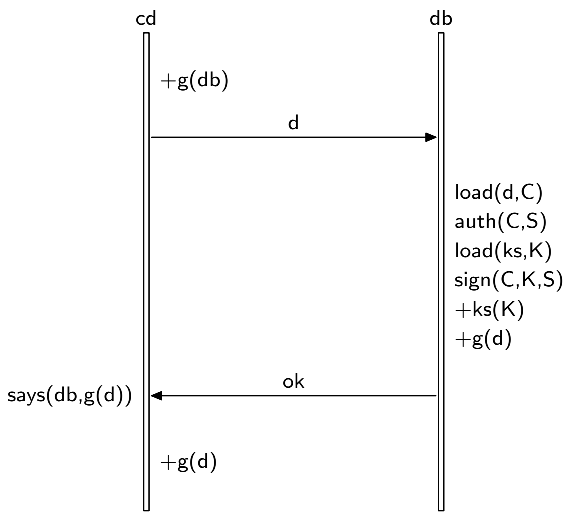

Figure 12: Authenticated Build: Display