Annotated Sequence Diagrams

Jonathan K. Millen and John D. Ramsdell

The MITRE Corporation
Bedford, MA USA

Abstract. An *annotated sequence diagram (ASD)* is a representation for a distributed logical computation in a system of locally communicating components, where communication between components is reliable and authenticated, but some components may have been compromised. We give a semantics for ASDs using multiset rewriting (MSR). Deduction rules specify how an ASD is compiled into a set of multiset rewrite rules. A Prolog program performs the compilation and interprets the MSR rules. With it, we can perform some rudimentary model checking to establish claimed properties of a system design or catch logical errors. We can also examine the consequences of malicious corruption of components. The approach is being applied to chain-of-trust arguments in a trusted platform design.

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 what a component believes about other components and its own data.

Our motivating application is 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". An adequate background on TPMs and supporting system architecture is beyond the scope of this paper, but we will give a small example of the kind of problem we address.

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 Definitions

An ASD is a pair (S, I) where S is a sequence of *steps* and I is a set of *inference rules.* 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. 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.

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

A message step from p to q is an expression $\mathsf{t}(p, q, \bar{x}, \phi)$ where p and q are principals such that $p \neq q$. In this step, $\bar{x}$ 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 $\mathsf{i}(p, \phi)$.

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 diagram in Figure 1 arises from the following step sequence:

$\mathsf{c}(p, f(X))$,
$\mathsf{t}(p, q, \langle X \rangle, f(X))$,
$\mathsf{t}(q, r, \langle p \rangle, \mathsf{true})$,
$\mathsf{c}(r, g(p))$,
$\mathsf{t}(r, q, \langle p \rangle, g(p))$,
$\mathsf{i}(q, g(p))$,
$\mathsf{i}(q, f(X))$

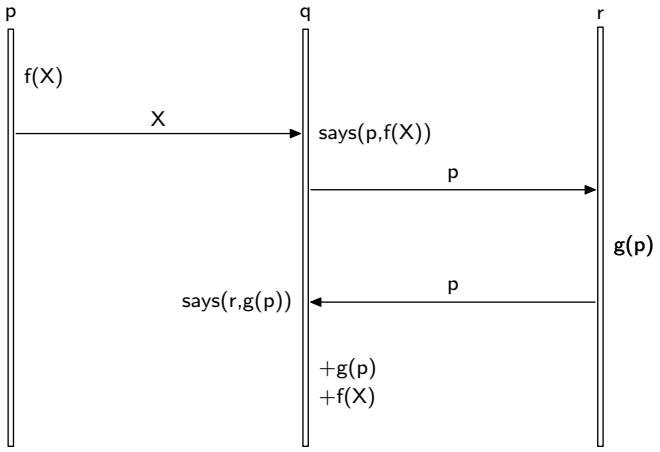

Fig. 1. Diagram, Trust Example

A step-sequence specification excludes some asynchronous behavior, since send and receive events are synchronized. If p sends a message to

q, and then receives a message from q, then q must have sent its message after receiving the one from p. However, the global ordering of steps is only partial, because our semantics allows for some concurrency across different components, when event ordering is not forced by message synchronization.

Our convention is to use capital letters for variables. When two principals share the use of a 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.

1.2 Beliefs and Assertions

The predicates appearing as annotations in the diagram are beliefs of the principal labelling the timeline. In the multiset semantics, an annotation $f(X)$ on the p timeline, arising from a computation, will become a fact $\mathsf{b}(p, f(x))$ with an explicit *belief* operator and some constant x instantiating X.

A message to q implying $f(x)$ will become a fact $\mathsf{b}(q, \mathsf{s}(p, f(x)))$, since the trustworthiness of the sender has yet to be established. (The operator s is expanded to says in the diagram for readability.) The identity of the sender is known, however, since communication is authenticated.

An annotation arising from an inference is shown with a preceding plus (+) in the diagram, as a convenient reminder. But the actual belief does not distinguish a computation from an inference.

There is a particular predicate symbol that shows up in our annotations: g. This is an *integrity* predicate. If $\mathsf{g}(p)$ is true, the principal p is honest and follows its role in the ASD. This integrity predicate is central to trust arguments. As we shall see in the discussion on semantics, it will be necessary, also, to support a complementary predicate n, such that $\mathsf{n}(p)$ means that the principal p is dishonest. A dishonest principal represents a corrupted or compromised component.

1.3 Trust Example Inferences

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 supposed to be a measurement agent that is able to determine this. Principal q trusts r implicitly.

The inference rule set I specified for this ASD has two rules:

- $\mathsf{b}(P, \mathsf{s}(r, \mathsf{g}(Q))) \rightarrow \mathsf{b}(P, \mathsf{g}(Q))$. That is, if P believes that r says that Q is good, then P believes that Q is good.
- $\mathsf{b}(P, \mathsf{s}(Q, A)), \mathsf{b}(P, \mathsf{g}(Q)) \rightarrow \mathsf{b}(P, A)$. 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 (with undesired results) when r is compromised.

The first rule justifies the inference by q of $\mathsf{g}(p)$. This conclusion is applied with the second rule to justify the next inference of $f(X)$. When we run the ASD checking tool, it confirms that these inferences can be obtained from the given rule set, and it also tells us what happens if components such as p and r are compromised. These results will be illustrated after we explain the ASD multiset rewriting semantics in the next section, and how it is implemented in the checking tool.

2 ASD Multiset Rewriting Semantics

As a system specification, an ASD can be simulated by a multiset rewrite (MSR) system. The MSR system has a state holding the beliefs of each principal. The state evolves according to the rewrite rules. The goal of executing such a system is to determine the reachable set of final states, that is, which beliefs are attainable to each of the principals.

2.1 The Multiset Model

In an MSR system, the state is a multiset of facts. A fact is a ground first-order atomic formula. The atomic formulas used in the ASD model are terms over the signature given in Section 2.2.

State transitions are expressed as rules in the following syntactic form:

$$\phi_1, \ldots, \phi_n \longrightarrow (\forall \bar{x})\psi_1, \ldots, \psi_m$$

where the quantification is optional and the ϕ_i and ψ_j are atomic formulas. All variables occurring in the ψ_j must occur either in some ϕ_i or in the vector $\bar{x}$. For convenience, quantified variables must not occur on the left.

A rule is enabled if there is a substitution σ from variables into data constants such that for each term ϕ_i on the left, $\phi_i\sigma$ occurs at least once in the multiset. If there is no quantifier, the rule is applied by deleting one occurrence of each left-side fact $\phi_i\sigma$ from the multiset, and, for each ψ_j on the right, adding one occurrence of $\psi_j\sigma$ to the multiset.

If there is a quantifier $(\forall \bar{x})$, the substitution σ must also map the variables x_i to data constants. The universe of data constants that may instantiate the quantified variables includes all constants, whether they occur in the multiset or are fresh data constants.

The MSR model used here is similar to, and motivated by, the model used in [4] for protocol analysis. However, we do not use the existential quantifier to produce fresh constants representing nonces. Our quantifier is needed to highlight the behavior of compromised principals in message steps. This usage is explained below in Section 2.6.

2.2 MSR Syntax

Figure 2 presents the signature used in expressing MSR facts. In this model, a fact is either

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

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 operations at the top level of facts. However, when we present the formalism below, we will revert to writing $f(\bar{x})$ rather than $\mathsf{a}(f, \bar{x})$ 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 variables. The association between values and variables is implicit in the order in which they appear.

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)	

Fig. 2. ASD Multiset Rewriting Signature

2.3 ASD Translation

An ASD specification is translated into multiset rewrite rules. A set of deduction rules in a structured operational semantics style is used below to specify this translation.

We think of the ASD specification as a program in a high-level language that is "compiled" into an intermediate language, as rewrite rules. The rewrite rules are interpreted on an abstract machine that can be emulated by Prolog or other means. The MSR representation is nondeterministic because of the nature of multiset rewriting and because of the particular rules that are generated for an ASD.

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

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

where $H : \mathsf{prin} \to \mathsf{nat}$ is the local state number and $\Delta : \mathsf{prin} \to \wp(X)$ is the local state context map, indicating which variables are defined for principal p. On the right is the transformed state information Δ' and H' and the set R of rewrite rules corresponding to 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 variables already defined.

The first deduction rule is essentially a recursion that reduces the translation of the step sequence to the translation of individual steps. 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.

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

Here, $s^\frown S$ is the concatenation of the step s to the sequence S. An empty step sequence transforms to an empty rule set.

2.4 Computation Step

In a computation step, a principal ensures that a predicate is true. Sometimes this causes variables to become defined. Recall that the compilation context Δ_p is the set of 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.

Treating a computation as a list lookup is not practical for a system implementation, for most predicates, but it works for abstract modeling purposes. Note that computational facts are global. A computation step just creates an explicit belief for a particular principal.

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

$$I \vdash \Delta, H, \mathsf{c}(p, f(\bar{x})) \twoheadrightarrow \Delta', H', \rho \tag{3}$$

where

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

Here we use the notation $\bar{z} \cup \bar{x}$ for a vector representing the union of the sets of variables represented by $\bar{z}$ and $\bar{x}$. The order turns out not to matter. The updated function $H' = H[p \mapsto H_p + 1]$ is identical to H except that $H'_p = H_p + 1$; this notation is used with Δ also.

2.5 Inference Step

In an inference step, a principal ensures that a predicate is true by finding an instantiation of an inference rule with that conclusion. The generated rewrite rule checks that the instantiation of the hypotheses are present in the multiset. If the principal is compromised, it does not check the predicate or update its beliefs.

$$\frac{(\bar{\psi} \to \phi) \in I}{I \vdash \Delta, H, \mathsf{i}(p, \phi\sigma) \twoheadrightarrow \Delta, H', \rho} \tag{4}$$

where

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

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

There is an important observation to be made in applying rule (4): there may be more than one inference rule and substitution that produce the step conclusion $\phi\sigma$. If so, several judgements may arise from (4), 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.

2.6 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 variable x that already occurs in the receiver's context, the rule permits a transition only when the ground values agree.

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

where

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

In the third (compromised-p) rewrite rule, the universally quantified variables in $\bar{y}$ are chosen such that $[\bar{y}] \cap [\bar{z}] = \emptyset$, and $|\bar{y}| = |\bar{x}|$. 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.

2.7 MSR Execution

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.

We expect that that a simulation or model-checking tool that exercises the rewrite rules will have some strategy for exploring the possible ways to instantiate quantified variables 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. This model is about trust and integrity. Keep in mind that messages are authenticated by assumption. Signatures may occur as arguments in predicates, and such predicates may appear as hypotheses for inferences, so there is still a way for digital signatures to affect conclusions about trust or other properties.

2.8 Prolog ASD Checker

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.

Generation of state sequences is handled with a very simple rewrite system. The multiset is a Prolog list of facts. There is a predicate `rule` relating a multiset `S1` to a possible next multiset `S2` by applying a rewrite rule. The `rule` predicate is used to find maximal state sequences from a given initial state. Backtracking permits all such sequences to be generated, since there are only a finite number from an ASD. Each rewrite rule ($L \rightarrow R$) becomes a clause of `rule`, and invokes a general `rewrite` predicate for updating the fact list.

For example,

```
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])]).

rewrite(S1,S2,L,R) :-
  subset(L,S1),
```

```
subtract(S1,L,S0),
append(R,S0,S2).
```

It is important for `subset` to backtrack and try all possible matches for L. We found that the built-in `subset` in SWI-Prolog stops at the first success, so we had to replace it with a custom predicate.

3 Analysis Examples

We now return briefly to the example in Figure 1 to consider the rest of its analysis, and then present a TPM-related example.

3.1 Trust Example Analysis

When the example diagram is run, the Prolog ASD tool confirms that a multiset with all the beliefs shown in the diagram is reachable. A maximal state sequence is called a *trace*. The tool displays the last state in the trace, and then continues to find additional traces if requested.

The standard initial state is equipped with an integrity fact $\mathsf{g}(p)$ for each principal p. The tool facilitates alternate initial states in which $\mathsf{g}(p)$ is changed to $\mathsf{n}(p)$ for any desired principals to be regarded as dishonest.

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 universal 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, with X still a variable. The final state is shown as:

q
says(p,f(X))
says(r,g(p))
-g(p)
f(X)

What is shown here is the output display rather than the actual belief facts. In the output display, inferences are not distinguished with +. However, the tool checks integrity beliefs against the integrity facts in the multiset, and marks a false belief with minus (-). Here, it has noted q's false belief in p's honesty. Any belief that follows a false belief should be considered suspect.

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.

Figure 3 shows a fragment 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 this 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 a trusted external entity.

The specification for this ASD (written in a Prolog syntax from which we have spared the reader) includes the crucial inference rule:

```
b(db,load(ks_hash_addr,A)),
b(db,readNVR(A,H)),
b(db,hash(K,H))
→ b(db,ks(K))
```

The ASD tool confirmed that enough inference rules had been provided to justify the final beliefs. 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

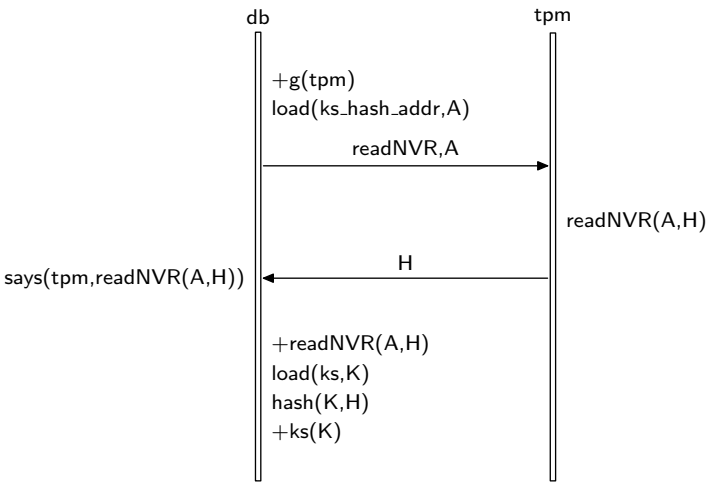

Fig. 3. Get Signing Key

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 is 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. The Prolog tool needs improvements to facilitate user interaction and automate aspects of property testing. We also anticipate a need to extend the present monotonic logic of annotations with the ability to update beliefs holding state information.

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`.