Document Logic: Risk Analysis of Business Processes through Document Authenticity

Shusaku Iida

School of Network and Information, Senshu University
2-1-1 Higashimita, Tama, Kawasaki, Kanagawa, 214-8580, Japan
Email: iida@isc.senshu-u.ac.jp

Grit Denker and Carolyn Talcott

Computer Science Laboratory, SRI International
333 Ravenswood Avenue, Menlo Park, CA 94025-3493, USA
Email: grit.denker@sri.com; carolyn.talcott@sri.com

Document Logic is a simple yet powerful framework to infer risks in business processes. We focus on flows of documents and build a set of inference rules based on document authenticity and a simple trust model. We have built a prototype of a system that checks document authenticity in Maude, an implementation of rewriting logic. Rewriting logic is expressive and general enough to define other specialized logics, like Document Logic. In our framework, a business process is modeled as a transition system. Our prototype takes a business process and an undesired situation as its input and outputs all the possible risks in the business process that could lead to the undesired situation.

Classification: D.2.2 (Software Engineering): Design Tools and Techniques; F.4.4 (Theory of Computation): Formal Languages; H.4.1 (Information Systems Applications): Office Automation-Workflow Management

Keywords: Formal methods, business process modeling, risk analysis, internal control, rewriting logic

1. INTRODUCTION

Business and risks are two sides of the same coin. In other words, there is no business without risk. However, unintended risks involved in a business may cause serious damage to all the stakeholders. In this paper, we propose a logical framework to analyze business processes.

A business process in general is a set of business entities together with their activities for certain business purposes. It is often designed by using a business process modeling language, such as, BPEL (TC, 2007) (Business Process Execution Language) or BPMN (Group, 2009) (Business Process Modeling Notation), or by using a standard modeling language, such as UML (Unified Modeling Language), flowcharts, and so on. We must choose an appropriate modeling language (or several languages) to represent the aspects in which we are interested. At the same time, it is important to choose an appropriate abstraction level. An important goal of our research is to provide

Manuscript received: 1 April 2010
Communicating Editor: Georg Grossmann

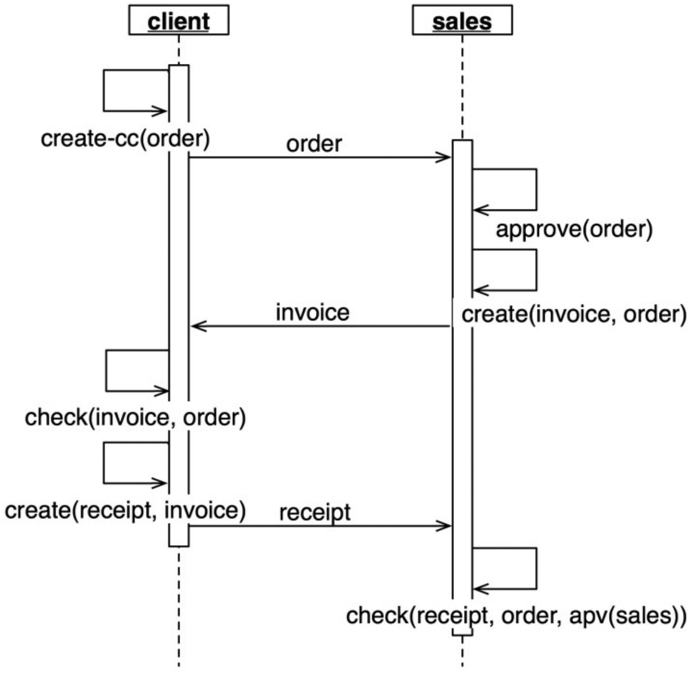

Figure 1: Simple Business Process Flow

a model and an analysis method that helps designers of business processes to formulate and analyze risks. Our model provides concepts, a language, and an abstraction level appropriate to capture the essentials of business processes, yet amenable to formal analysis. In this paper, our interest is in the flow of a business process, i.e., *who does what*, *when*. We use the phrase *Business Process Flow* (abbreviated to BPF) to clarify that we are focusing on the flow of a business process.

1.1 Business Process Flow

Figure 1 shows a simple and small BPF example. It is modeled by using a UML Sequence Diagram. We have two business entities, client and sales, and messages are passed between these entities. A business entity can be a division of a company, a worker, a client, a computer system, and so on. In this paper, we simply use the word *division* to represent business entities. We restrict the elements of the Sequence Diagram as follows to make the diagram specific to our purpose:

- Each object is a division.
- Each message passed between divisions is a document.
- Each message passed from a division to itself is a special activity either of the type creation message or control message.

The messages of type creation message are *create*(d) for creating a document d, *create*(d_1, d_2) for creating a document d_1 from a document d_2, *create-cc*(d) for creating a document d with a carbon copy, and *create-cc*(d_1, d_2) for creating a document d_1 from document d_2 with a carbon copy.

Each message of type control message consists of two categories: *check* and *approve*, which means that we can check and approve a document respectively. If a document is checked or approved, then it is marked by "something", e.g., a signature or a seal, to record that it is already checked or approved. We call this "something" *evidence of a check* (*evidence* for short). An evidence of a check contains information about which division has checked the document.

The category check consists of the following four messages. These messages are all for checking a document d_1 with document d_2, but have different evidences:

- $check(d_1, d_2)$ for checking a document d_1 with document d_2,
- $check(d_1, e_1, d_2)$ for checking a document d_1 that has evidence e_1 with document d_2,
- $check(d_1, d_2, e_2)$ for checking a document d_1 with document d_2 that has an evidence e_2 and
- $check(d_1, e_1, d_2, e_2)$ for checking a document d_1 that has evidence e_1 with document d_2 that has an evidence e_2.

If a document d is checked by a division v, then an evidence $ch(v)$ is written in d.
The category approve consists of the following messages:

- $approve(d)$ for approving a document d and
- $approve(d, e)$ for approving document d which has an evidence e.

If a document d is approved by a division v, then an evidence $apv(v)$ is written in d.

1.2 The World of Business Process

Figure 1 represents a simple BPF. There are two business entities: client and sales. However, in real business situations, there would be several clients and there may be several office workers in the sales division. In addition, there may be several attackers planning to do illegal activities. So, the world of business process can be considered as a soup of business entities (and attackers). These business entities are working concurrently and they are communicating with each other.

A business process like Figure 1 shows only a session in this world. When we analyze a BPF, it is important to clarify which of the following is our concern:

- analysis of a single session or
- analysis of multiple sessions.

For the former case, we are considering only one session and business entities that are related to the session. We also consider an unbounded number of attackers. Let us call this model the *micro view* of a BPF. For the latter case, we are considering unbounded numbers of business entities and attackers, and unbounded numbers of sessions between them. Let us call this model the *macro view* of a BPF. In addition to these two views, we can consider a wider view that we call the *enterprise view* of BPFs in a company. The enterprise view provides a model for all the business entities and BPFs in a company.

The meaning of risk analysis differs among these views. In the micro view model, we only consider risks within a session. In the macro view model, we also consider risks that will be caused by interference of different sessions of one BPF. In the enterprise view model, we are interested in whole activities that happen in a company. In this case we also consider interference that may arise between different BPFs.

In this paper, we discuss the micro and the macro view. We do not discuss the enterprise view, since it requires new analysis methods to manage the complexity of interactions.

1.3 Risks in Business Process Flow

There are many kinds of risks in business. The risks we can discover in a BPF depend on how we model the BPF. Risks such as natural disasters, economic crises, and personal problems are not our concern. Our concern is to discover the possibilities of illegal activities, such as document forging, embezzlement, and illegal selling. So, we focus on information flows among organizations. For example, in Figure 1, the following risks could be of concern:

- The sales division may receive a forged order.
- An office worker in the sales division may forge an order.
- An office worker in the sales division may forge an invoice.

These risks come from illegal documents. Our analysis focuses on the authenticity of a document.

Risks are not always a bad thing in business. Most companies take some risks and those risks are the source of their business. So, we have to clarify which risks are expected and which are not, and design controls to prevent unexpected risks from occurring. Document Logic only captures unexpected risks that derive from specified activities. Other activities that are not specified in our model, even though they might represent a risk, are risks that are expected. For example, we assume that signatures cannot be forged, which means we are ready to take risks caused by that. As a consequence, in our Document Logic model there is no rule that corresponds to signature forging. On the other hand, if we do not set controls properly, then unexpected risks remain in a BPF, and those risks are our concern.

An unexpected risk should be controlled by activities that are not directly related to the business purpose. So, it is important to minimize a set of controls to prevent a risk. Therefore, we immediately face questions such as the following:

- Do the controls cover risks sufficiently?
- Are the controls efficient?

In this paper, we propose a framework that allows users to infer the effectiveness of the controls they set.

1.4 Trust Model

Risks depend on trust. If we do not trust any division, then anything can happen. On the contrary, if we trust all the divisions, then there is no risk at all within our context. So, we have to decide which divisions are trustworthy and which are not. Also, we assume that the numbers of illegal activities happening in a BPF are bounded by a certain number. This is done to reduce the complexity of problems and make models more amenable to analysis. We have modeled this by two notions: *untrusted division* and *illegal activity bound*.

1.5 Related Work

This research can be seen as a bridge between business process analysis and protocol analysis. There are many studies for business process analysis, such as Van der Aalst and de Graaf (2002) and Barth *et al* (2007). These studies mainly focus on how to prove the correctness of a business process only with proper players. On the other hand, many studies on protocol analysis such as Dolev and Yao (1983); Thayer Fa´brega *et al* (1998) and Denker *et al* (1998) assume malicious players who try to attack the proto- col. We regard a BPF as a protocol between different divisions and assume attackers who are trying to do something illegal.

Risk analysis of a business process is required in many different situations, such as designing a business process, evaluating an important business process, and auditing business processes of a company. One should select an appropriate analysis method for these different situations. For some situations, a model checking technique provides a good solution, and maybe for other situations simpler methods can be used. However, there is not much research providing a good foundation that supports several analysis methods. We provide several analysis methods that a user can choose according to a situation. To achieve this, we need a base-level language to specify a BPF, and a meta-level mechanism to implement different analysis methods. We show how we can achieve this in Section 5.

Our work is mostly inspired by Escobar *et al* (2006), and our analysis approach follows the ideas presented in that paper.

1.6 Structure of this Paper

In Section 2, we give a basic definition of rewriting logic. In Section 3, we define our model of BPFs such as a document, a division, a BFP state. In Section 4, we give the rules of our logic and define the relation between a BPF and the rules. Section 5 explains our analysis method of the micro view, that is, analyzing risks for a given BPF. In Section 6 we extend Document Logic to fit the macro view model of a BPF that has several sessions. In Section 7 we present two analyses that highlight the differences between micro and macro view models: (1) backward reachability analysis of the micro view model, and (2) forward reachability analysis of the macro view model. We conclude with final remarks in Section 8.

2. PRELIMINARIES

We use *rewriting logic* (Meseguer, 1992) to formalize our framework. Here, we provide definitions used in this paper. We encourage readers to see Meseguer (1992; 1998) for more details.

Rewriting logic is a powerful and general logic that can be used for implementing other logics. It is an extension of membership equational logic (Meseguer, 1998). Its operational semantics is given by a *term rewriting system* (TRS), and its model-oriented semantics is given by categories with algebraic structure. Here, we mainly focus on the operational semantics.

A *term* is constructed by function symbols and variables. *Sorts* are like types in a programming language. They, together with function symbols, are used to define well-formed terms. So, if we have two sorts A and B, a function $f : A \rightarrow B$, and constants $a : \rightarrow A$ and $b : \rightarrow B$, then $f(a)$ is a well-formed term of sort B but $f(b)$ is not well-formed. However, if we add a sort inclusion $B < A$, then $f(b)$ is also a well-formed term. We can define axioms for a function, such as associativity, commutativity, idempotency, and identity. We call these axioms *equational attributes* of the given function. Suppose we have a function $f : Nat \times Nat \rightarrow Nat$ with commutativity, then $f(f(3,4),5)$ is equal to $f(5, f(4,3))$. A *signature* is a triple $(S, \Sigma, <)$ where S is a set of sorts, Σ is a set of sorted function symbols (such as $f : s_1 \times s_{n-1} \rightarrow s_n$, where $s_1 \dots s_n \in S$), and $<$ is an order (sort inclusions) on S. A set of well-formed terms (called Σ-term) constructed from a signature $(S, \Sigma, <)$ is represented as T_Σ. We can consider a set of well-formed terms constructed from $(S, \Sigma, <)$ and variables X that is also sorted with S. In this case, we represent the set as $T_{\Sigma(X)}$, and we call an element of $T_{\Sigma(X)}$ a *pattern*. A ground term is a term that does not contain any variable. A *substitution* is a sort preserving function $\theta : X \rightarrow T_{\Sigma(Y)}$. We can extend a substitution homomorphically to terms as $\overline{\theta} : T_{\Sigma(X)} \rightarrow T_{\Sigma(Y)}$. A *subterm* of a term t is a subtree of t and is also a well-formed term. We represent a subterm of t by t / π where π is a position of the subterm and $t[t']_\pi$ is the term t with the subterm at position π replaced by t'. We can match a ground term t to a pattern p with a set of variables X if there exists a substitution $\theta : X \rightarrow T_\Sigma$ such that $\overline{\theta}(p) = t$.

An *equational theory* is a pair $(\Omega, E \cup A)$, where Ω is a signature (i.e., $(S, \Sigma, <)$), E is a set of Σ-*equations*, A is a set of equational attributes. A Σ-equation is a possibly conditional equation as follows:

$t = t'$ *if* $u_o = v_0 \wedge \ldots \wedge u_n = v_n$

where $t, t', u_o \ldots u_n, v_0 \ldots v_n \in T_{\Sigma(X)}$. When the list of equations for the condition is empty, we have an unconditional Σ-equation. The term t is called the *left hand side* of the equation (*lhs*) and t' is called the *right hand side* of the equation (*rhs*).

If we replace the symbol = in Σ-equations E with the symbol →, then we get a set of rewrite rules R_E. We can *rewrite* a ground Σ-term t to another term by using R_E and a set of equational attributes A. A step of rewriting is defined as follows:

1. Find a rule $l \to r$ in R_E such that the pattern l matches t / π modulo A with substitution $\overline{\theta}$.
2. If the matched rule is conditional, then check for all the conditions (we explain how to check conditions later).
3. If all the conditions hold, then we have a rewrite step $t \to t[\overline{\theta}(r)]_\pi$.

For example, suppose that we have the equational theory with the set of sort {*Nat*}, the set of variables $\{n,m\}$, the set of function symbols $\{0 :\to Nat, s: Nat \to Nat, +: Nat \times Nat \to Nat$, the set of equations $\{n+s(m) = s(n+m)\}$, and + as associative and commutative. Then, we can rewrite the ground term $s(s(0))+0+0$ to $s(0+s(0))+0$, because we can match the $s(s(0))+0$ to the pattern $n+s(m)$ (modulo commutative) with the substitution $\{n \mapsto 0, m \mapsto s(0)\}$. With R_E and the rewriting step, we can define the *equational rewriting relation* $\to_E$ such that $t \to_E t'$ if t can rewrite to t' by using R_E. If a term cannot be rewritten further, then it is called a *normal form*. If there is no infinite sequence of rewrites for any term, then R_E is called *terminating*. Suppose u can rewrite to v and also to v'; if we can rewrite both v and v' to u', then we call R_E *confluent*. We call R_E *convergent*, if it is terminating and confluent. If R_E is convergent, then there exists a unique normal form called *canonical form* for each $t \in T_\Sigma$. Suppose we have a convergent set of rewrite rules R_E. If for each rule $(l \to r \in RE)$ $Var(r) \subseteq Var(l)$, then we can use $\to_E$ to prove an equality $t = t'$ by computing the canonical form of t and t' such that $Can_E(t) = Can_E(t')$ (where the function *Var* gives us a set of variables in a term, and Can_E gives the canonical term of t by $\to_E$). Conditions of a Σ-equation are checked also by checking $Can_E(u_0) = Can_E(v_0) \ldots Can_E(u_n) = Can_E(v_n)$.

A *rewrite theory* is a triple $(\Omega, E \cup A, R)$, where $(\Omega, E \cup A)$ is an equational theory (E is a set of equations and A is a set of equational attributes) and R is a set of rewrite rules. A rewrite rule is $l \to r$, where l is a pattern and r is a Σ-term (A rewrite rule is either conditional or unconditional. Here, we only consider the unconditional case). From the computational point of view, R_E should be convergent; however, R does not have to be (but still should have $Var(r) \subseteq Var(l)$ for every rule $l \to r$ in R and we also need R to be coherent (Meseguer, 1992)). By using R_E, we compute equality between terms, and by using R, we compute reachability between terms. A ground term t can be rewritten by using $R_E \cup R$ and A as follows:

1. Find a rule $l \to r$ in R such that the pattern l matches $Can_E(t)/\pi$ modulo A with substitution $\overline{\theta}$.
2. Then, we have a rewrite step $t \to Can_E(t[\overline{\theta}(r)]_\pi)$.

If t rewrites to t' in one step, then we say t and t' are in one-step rewrite relation and denote it as $t \to^1 t'$. We denote the reflexive and transitive closure of $\to^1$ as $\to^*$. We say a term t' is *reachable* from a term t if $t \to^* t'$.

If two terms $t, t' \in T_{\Sigma(X)}$ can be equal by using the same substitution $\theta : X \to T_{\Sigma(Y)}$ as $\overline{\theta}(t) = \overline{\theta}(t')$, then we call θ a *unifier*. A unifier σ is *more general* than θ if there exists a unifier η such that $\theta = \sigma\eta$ ($\sigma\eta$ is a unifier that can be obtained by composing σ and η). A unifier σ is the *most general unifier* (mgu) if for every unifier θ of t and t' σ is more general than θ. If no function in a rewrite theory has an equational attribute and terms t and t' have a unifier, then there exists a unique mgu. (This is true in the many-sorted case (which does not have any order on sorts); for order-sorted theories there can be several unifiers, even when no equational attribute is used.). If we have equational attributes, then we cannot have a unique mgu but instead we may have a *complete set of unifiers* (CSU), which means that for every unifier θ of t and t' we can find σ in CSU such that σ is more general than θ. It is well known that if we have a function only with associativity for its equational attribute then CSU can be an infinite set. However, in the case of having a function with commutativity alone or associativity with commutativity, CSU is finite.

Narrowing is another way of computing reachability by using $R_E \cup R$ of a rewrite theory $(\Omega, E \cup A, R)$. A step of narrowing $t \in T_{\Sigma(X)}$ is defined as follows (here we give the definition only for the unconditional case):

1. Find a rule $l \rightarrow r$ in $R_E \cup R$ such that there exists a position π in term t and a unifier θ (which is an element of CSU of l and t / π) such that $\overline{\theta}(l) = \overline{\theta}(t / \pi)$.
2. Then we have a narrow step $t \rightsquigarrow \overline{\theta}(t[\mathrm{r}]_{\pi})$.

In a rewriting step, we cannot use a rule $l \rightarrow r$ such that $Var(l) \not\subseteq Var(r)$, but in a narrowing step there is no such restriction.

2.1 Multisets and Lists

In this paper, we heavily use sets and lists for basic data structures. We use a multiset instead of using a set, because of the computational cost. A multiset is defined by a function _ _:*MSet* × *MSet* →*MSet* that is associative, commutative, and has an identity element *empty*. Multisets allow duplication of elements, so we do not have idempotency. The function _ _ is a special notation for multiset union. The use of empty syntax for multiset union follows traditional mathematical convention. For example, if *A* and *B* are both multisets then *A B* is also a multiset. We have associativity and commutativity, so (*C B*) *A* is equal to *A B C*. Also, we have identity for *empty*, so *A empty* is equal to *A*. When we want to specify the type of elements of a multiset, we can define subsort inclusion, including each element as a singleton multiset. For example, if we want multisets of natural numbers, then we add the subsort relation *Nat* < *MSet*. The following are examples of multisets of natural numbers:

1 5 10,   *1,*   *empty,*   *4 4 4,*   *5 empty*.

A list is defined by a function _;_: List List → List that is associative, and has an identity element nil. A list of natural numbers can be defined by Nat < List. The following are examples of lists of natural numbers:

9; 3; 8,   *2,*   *nil,*   *1; 1; 1, 3; nil*.

3. MODELING BUSINESS PROCESSES FLOW

We define several components of the BPF model, such as document, division, and cabinet. We also define how we can execute a BPF.

3.1 Basic Entities

Documentation is an essential activity in any business. In other words, every business activity should be recorded. We focus on two information aspects of business documents, authenticity and evidence history, to trace the flow of a document in a BPF.

- Authenticity of a document means whether or not the document is real. If it is not real, then it means that the document is forged.
- A document can be checked against another document, and also a document can be approved. These activities are recorded as an evidence history in the document.

In the real world, checks and approvals are usually recorded by the signature of the person who checks (or approves) the document. In our model, we do not care which individual checked the document, but we do care which document was used to check the document. Also, when approving a document, we record only the division that approves the document.

Definition 1 (*Evidence of a check and evidence history*). An *evidence of a check* is constructed by the following functions:

$$ch : DocType \to Evidence \qquad apv : Div \to Evidence$$

where *DocType* is a finite set of document types, *Evidence* is a set of evidences, and *Div* is a finite set of divisions. An *evidence history* is a set of evidence. The set of all evidence histories is denoted *EvidenceHistory*.

We use set instead of list because of an implementation issue. Specifically, the analysis tool we use, Maude, only supports use of (multi)set type axioms for unification and narrowing. If we need to reason about ordering between evidence items, this could be done by adding suitable annotations to the evidence.

Definition 2 (*Document*). A *document* is constructed by the following function:

$$doc: DocType \times Bool \times EvidenceHistory \to Doc$$

where *Bool* is used to represent authenticity of the document. If the authenticity of a document is true, the document is real. If it is false, the document is forged. An element of *Doc* represents an abstraction of a document, in the sense that we are not concerned with its content.

Example 1 Suppose that we have a set of document types $\{order, invoice\}$, then $doc(order, true, empty)$ is an *order* that is not forged, and this document has not yet been checked with any document. The following is an *invoice* that is forged and has been checked with the *order*:

$$doc(invoice, false, ch(order))$$

A *division* is an entity that sends and receives a message (document). For example, client, sales division, shipping division, accounting division, and so on are considered to be divisions. Each document is stored in a division; we call this the *document's location*, and we call a set of document's locations a cabinet.

Definition 3 (*Document's location and cabinet*). A *document's location* is constructed by the following function:

$$in : Doc \times Div \to DocLoc$$

where *DocLoc* is the set of all documents' locations. A *cabinet* is a finite set of documents' locations. We denote the set of all cabinets *Cabinet*.

Example 2 Suppose that we have a set of document types $\{order, invoice\}$ and a set of divisions $\{client, sales\}$; then $in(doc\ (order, true, empty))$ is a document's location, which means that *client* has the *order*. The following is a cabinet:

in(*doc* (*order*, *true*, *empty*), *client*) *in*(*doc* (*invoice*, *false*, *empty*), *sales*)

where we have two documents' locations in the cabinet (recall the definition of set in Section 2.1).

3.2 State Space of a Business Process Flow

Each activity of a division is modeled as a strand (Thayer Fábrega *et al*, 1998). A strand is a list of messages. There are three message types: input/output, document creation, and control message.

Definition 4 (*Input/output message*). An *input message* is constructed by the function *rec*: *DocType* × *Div*→*SMsg*, where *SMsg* denotes the set of all messages. An *output message* is constructed by the function *snd*: *DocType* × *Div*→*SMsg*

A document can be created by either using information in another document, or without using any other information. So, we have two types of document creation for messages. Similarly, a document can be created with a carbon copy (cc).

Definition 5 (*Document creation for messages*). A *document creation message* is constructed by the following functions:

create:*DocType*→*SMsg*   *create*:*DocType* × *DocType*→*SMsg*

The latter function represents a document having the document type of its first argument, but which is created from its second argument. A *document creation with cc message* is constructed by the following functions:

create-cc:*DocType*→*SMsg*   *create-cc*:*DocType* × *DocType*→*SMsg*

The latter function represents a document of the type of its first argument, which is created (with cc) from its second argument.

There are six instances for control message type: four check messages and two approve messages.

Definition 6 (*Check message*). *Check messages* are constructed by the following functions:

check: *DocType* × *DocType*→*SMsg*
check: *DocType* × *Evidence* × *DocType*→*SMsg*
check: *DocType* × *DocType* × *Evidence*→*SMsg*
check: *DocType* × *Evidence* × *DocType* × *Evidence*→*SMsg*

Basically, these messages are for checking a document specified by the first argument (we name the document *target document*) with a document specified by the second (in the case of the first and third message) or third (in the case of the second and fourth message) argument (we name the document *reference document*). Both target and reference document may have evidence history. Evidence is either the approval through a division or a check against another document. Thus, there are four different formats for the check message: The first message format is for checking a target document with the reference document and neither of them has evidence. The second and the third messages are for checking a target document with a reference document where one of them has evidence. The last message is for checking the target document with the reference document and both documents have evidence history as specified as the second and the fourth argument, respectively.

Definition 7 (*Approve messages*). *Approve messages* are constructed by the following functions:

$approve: DocType \rightarrow SMsg \qquad approve: DocType \times Evidence \rightarrow SMsg$

The second message means that to approve this document, the document should have the evidence specified by the second argument.

A strand is a list of messages. This structure specifies in which order the messages are supposed to be sent and received. This list is divided into two parts using the symbol $|$. Messages to the left of $|$ are past messages, meaning messages that have been already processed (i.e., messages that have been sent or received), and messages to the right of $|$ are future messages that still must be processed (i.e., message that must be sent or received). Thus, $|$ represents the current position in the execution of the strand.

Definition 8 (*Strand and current position*). A *strand* is a triple (v, L_1,L_2), where v is a division, and L_1 and L_2 are both lists of messages. We denote (v, L_1,L_2) as $v[L_1 \mid L_2]$. We call "$|$" the current position of $v[L_1 \mid L_2]$.

Example 3 Suppose that we have a set of divisions $\{client, sales\}$ and a set of document types $\{order, invoice\}$; then

$client\ [create(order) \mid snd(order, sales); rec(invoice, sales)]$

is a strand representing the client activity in Figure 1 (recall the definition of list in 2.1). It describes that the *client* has created an *order* and it will send *order* to *sales* and then receive *invoice* from *sales*.

Definition 9 (*Initial position and final position of a strand*). A strand is in its *initial position* if the first part of the list of messages is *nil*. A strand is in its *final position* if the second part of the list of messages is *nil*.

Before we define the state space of a BPF, we have one more concept to define. In our model, risks depend on our trust in divisions. This is modeled by a set of divisions that we do not trust.

Definition 10 (*Untrusted set*). An *untrusted set* is a set of divisions that we do not trust.

Now, we are ready to define the state space.

Definition 11 (*State*). A *state* of a BPF is a 4-tuple (S,C,U,n), where S is a set of strands, C is a cabinet, U is an untrusted set, and n is a natural number representing the illegal activity bound.

Example 4 A state of the BPF shown in Figure 1 can be modeled as follows:

$((client[create(order) \mid snd(order, sales); rec(invoice, sales)]$
$sales[nil \mid rec(order, client); create(invoice, order); snd(invoice, client)])$
$(in(doc(order, true, empty), client)), (sales), 2))$

This means that we have two strands for *client* and *sales*, and that *client* has created a document *order* and sales is waiting to receive it. In the cabinet, we can find *doc*(*order*, *true*, *empty*) in client. We do not trust the sales division and we allow as many as two illegal activities.

3.3 Execution

Execution of a BPF is performed by passing messages, and is defined by a set of rules in our model (the actual Document Logic rules are given in Section 4). A rule is modeled by a rewrite rule in rewriting logic.

Definition 12 (*Document Logic rule*). A rule of *Document Logic* is a rewrite rule of rewriting logic of the form $SP \rightarrow SP'$, where SP and SP' are both terms denoting patterns for states of a BPF. We denote a set of Document Logic rules as R_{DL}.

Usually, we begin execution at an initial state, and the current positions of the different strands gradually move from left to right until they reach a final state or there is no rule to be applied any more. In other words, we use rewriting to reach a final state from an initial state. We call this execution a *forward execution* of a BPF.

Definition 13 (*Initial state and final state*). Suppose that we have a state (S,C,U,n). A state is called an *initial state* if all the strands in S are in their initial position and C is empty. A state is called a *final state* if all the strands in S are in their final position.

If the rules have nondeterminism, that is, if we can apply several rules to a state, we may have several paths from an initial state to (possibly many) final states. We may also have several initial states, because we allow an unlimited number of attackers to participate in a BPF.

Instead of using a forward execution, we can start from a final state and use the rules in a backward way to search for an initial state. We call this execution a backward execution of a BPF.

Definition 14 (*Backward rule*). Suppose we have a Document Logic rule $l \rightarrow r$. The corresponding *backward rule* is $r \rightarrow l$. If we have a set of Document Logic rules R_{DL}, then we denote the set of corresponding backward rules as $\hat{R}_{DL}$. Note that in R_{DL} rewriting proceeds from initial toward final states, while in $\hat{R}_{DL}$ rewriting proceeds from final toward initial states.

In the following, we will present all rules as forward rules and indicate when we use them in forward fashion or in backward fashion for different analyses.

4. DOCUMENT LOGIC

Document Logic rules are specified as forward rules. Forward rules describe state transitions going forward in time in the sense that the left side of the rule describes the predecessor state, and the right side of the rule describes the successor state that can be reached from the preceding state. If one were to swap left and right sides of a rule, a forward rule becomes a backward rule. In a backward rule, the left side of the rule describes the successor state that can be reached from the preceding state described in the right side of the rule. In that sense, backward rules describe the evolution of systems states going backwards in time. The decision to formulate the state transitions of a system using forward or backward rules depends on the kind of analysis methods one wants to use. In Section 5 we will discuss two types of analyses of state transitions supported by Maude: reachability analysis using forward execution and reachability analysis using backward execution. We will see that for finding attacks in the micro view, reachability using backward execution is more expressive. Backwards narrowing is a search strategy that allows one to consider an unbounded number of initial states (with different numbers of sessions, attacks and so on) at once. This contrasts with model checking and forward search that starts with a concrete initial state and looks for one attack. Backward search can look for multiple attaches if they are reachable. Of course, one trades looking for many initial states in backward search for starting with a fixed attack pattern.

However, to ease readability, we will present all rules in forward fashion and point out for each analysis whether we use forward rules or the corresponding backward rules. Thus, in the context of analysis in the micro view, while we represent the BPF with forward rules, the analysis is done with the corresponding backward rules. In the macro view analysis, we will present the BPF using forward rules and use these rules in forwards reachability analysis to manage state space explosion by focusing on a fixed initial state.

Definition 15 (*Document Logic*). A *Document Logic* is a triple (Σ, A, R), where Σ is the set of function symbols defined as in Section 3, A consists of equations used only as equational attributes (i.e., associativity, commutativity, and identity), and R is either R_{DL} or $\hat{R}_{DL}$. Depending on analysis we use R_{DL} or $\hat{R}_{DL}$ (see Section 5 for analysis).

From now on, we assume the following variables: We use uppercase letters for sets and lists, and lowercase letters for other data; S for *StrandSet*, C for *Cabinet*, U for *Untrusted*, n for natural numbers, and H for *EvidenceHistory*; $v, v_1, \ldots$ for *Div*; $t, t_1, \ldots$ for *DocType*; $k, k_1, \ldots$ for *Evidence*; $b, b_1, \ldots$ for *Authenticity*; $ML, ML_1, \ldots$ for list of messages.

4.1 Proper Operations

We define "proper" operations for documents: *create*, *create-cc*, *snd*, *rec*, *check*, and *approve*. We mean proper in the sense that these are legal operations in the business process.

The *create* operation has two types: simply create a document, and create a document from another document.

Definition 16 (*Create-1*). The rule *create-1* is

$$((v[ML_1 \mid create(t); ML_2]S), C, U, n)$$
$$\rightarrow((v[ML_1; create(t) \mid ML_2]S), (in(doc(t, true, empty), v)\ C), U, n)$$

This rule can be read as follows: "If there is a strand whose current position is right before creating a document of type t, then it can transition into a new state in which the strand's current position is right after creating the message and we can find that document which is not forged and has an *empty* evidence history in the cabinet." Note that the strand is for the division in which the document is located (because we use the same variable v for the strand and for the document's location).

Definition 17 (*Create-2*). The rule *create-2* is:

$$((v[ML_1 \mid create(t_1, t_2)\ ; ML_2]S), (in(doc(t_2, b, H), v)\ C), U, n)$$
$$\rightarrow((v[ML_1; create(t_1, t_2) \mid ML_2]S), (in(doc(t_1, b, empty), v)\ in(doc(t_2, b, H), v)\ C), U, n)$$

The rule *create-2* says that if there is a strand whose current position is right before creating document t_1 from document t_2, and if the document of type a t_1 has not yet been created, then we can find in the next state both documents of type t_1 and t_2 with the same authenticity in the same document's location. The point is that we can create a real document only from a real one and we will create a forged document from a forged one even if we do not intend to forge a document. This rule expresses our assumption that in the real world a person who processes a document cannot change its validity from true to false or vice versa.

Definition 18 (*Carbon copy*). A *carbon copy* of a document is constructed by the function cc: $Doc \rightarrow Doc$.

Definition 19 (*Create-cc-1*). The rule *create-cc-1* is:

$$
\begin{aligned}
&((v[ML_1 \mid \mathit{create\text{-}cc}(t);\, ML_2]S),\, C,\, U,\, n)\\
&\rightarrow((v[ML_1;\, \mathit{create\text{-}cc}(t) \mid ML_2]S),\\
&\quad(in(doc(t, true, empty), v)\ in(cc(doc(t, true, empty)), v)\ C),\, U,\, n)
\end{aligned}
$$

The rule *create-cc-1* is almost the same as *create-1*, except that this is for creating a document and its carbon copy simultaneously. The original document and its carbon copy are essentially the same, except that the carbon copy is marked as such. Carbon copies are special in that an attacker never forges carbon copies.

We chose to model the create carbon copy message without indicating the division to which the carbon copy is sent, but it would be possible to add a parameter keeping track of the carbon copy. The main reason we decided against tracking the carbon copy is that it is not relevant for enabling new illegal activities, since the attacker cannot forge a carbon copy.

We omit the rule *create-cc-2*, which corresponds to create carbon copy for *create-2*.

Sending a document (*snd*) and receiving it (*rec*) happen synchronously, defined as follows:

Definition 20 (*Send*). The rule *send* is:

$$
\begin{aligned}
&((v_1[ML_1 \mid snd\ (t, v_2);\, ML_2]\ v_2\ [ML_3 \mid rec(t{,}v_1);\, ML_4]\ S),\, (in(doc(t, b, H), v_2)\ C),\, U,\, n)\\
&\rightarrow((v_1[ML_1\ ;\, snd\ (t, v_2) \mid ML_2]\ v_2\ [ML_3\ ;\, rec(t{,}v_1) \mid ML_4]\ S),\, (in(doc(t, b, H), v_2)\ C),\, U,\, n)
\end{aligned}
$$

We recall Definition 6, which defines four different messages to check a document with another. Accordingly, when writing rules for checking documents, all message formats need to be considered, which results in several rules for checking a document that are somewhat similar. Thus, in the following we define rules for checking documents for two message formats: (1) checking a target document with a reference document, neither of them having evidence, and (2) checking a target document that has evidence history with a reference document that has evidence history.

We can check a document with another document only if their document's locations are the same and they have the same document authenticity. It is natural to think a real document can be checked with another real document. In addition to that, a forged document can be checked with another forged document. To explain why we need the latter case, let us consider the following situation. Assume that we have two documents d_1 and d_2 and we want to check d_1 with d_2. If d_2 is forged, then it is possible that someone wants to trick the checker of d_1 and make up the contents of d_2. In this case, both d_1 and d_2 are forged. We are looking for risks, so if there is a possibility of illegal activity, we should make it traceable.

If the check is passed, then we add an evidence to the evidence history of the document (in the real world this can be seen as writing down the signature of the person who checks the document), so that we can know whether or not a document is checked.

Definition 21 (*Check-1*). The *rule check-1* is:

$$
\begin{aligned}
&((v[ML_1 \mid check\ (t_1, t_2);\, ML_2]\ S),\, (in(doc(t_1, b, H_1), v)\ in(doc(t_2, b, H_2), v)\ C),\, U,\, n)\\
&\rightarrow((v[ML_1\ ;\, check\ (t_1, t_2) \mid ML_2]\ S),\, (in(doc(t_1, b, (ch(t_2)\ H_1)), v)\ in(doc(t_2, b, H_2), v)\ C),\, U,\, n)
\end{aligned}
$$

We can check a document by using a carbon copy. The corresponding rule is exactly the same as the rule check except that we check a document with a carbon copy $cc(doc(t_2, b, H_2))$ instead of $doc(t_2, b, H_2)$. We omitted showing the rule.

As mentioned earlier, we will also provide the rule for checking a target document that has evidence history with a reference document that has evidence history. The corresponding check message is defined as *check*: $DocType \times Evidence \times DocType \times Evidence \rightarrow SMsg$. This check can be used for the situation that a document d_1 should be checked with d_2 and the checker should make sure that d_1 has an evidence k_1 that may be a signature of the manager of the checker, and also d_2 has an evidence k_2 that may be a signature of the other division's manager.

Definition 22 (*Check-7*). The rule *check-7* is:

$$\begin{aligned}&((v[ML_1 \mid check\ (t_1, k_1, t_2, k_2);\ ML_2]\ S),\\ &(in(doc(t_1, b, (k_1, H_1)), v)\ in(doc(t_2, b, (k_2, H_2)), v)\ C), U, n)\\ &\rightarrow((v[ML_1\ ;\ check\ (t_1, k_1, t_2, k_2) \mid ML_2]\ S),\\ &(in(doc(t_1, b, (ch(t_2)\ k_1\ H_1)), v)\ in(doc(t_2, b, (k_2, H_2)), v)\ C), U, n)\end{aligned}$$

To approve a document, we add an evidence $apv(v)$ to the evidence history of the document, where v is the division that the person who gave the approval belongs to.

Definition 23 (*Approve-1*). The rule *approve-1* is:

$$\begin{aligned}&((v[ML_1 \mid approve(t)\ ;\ ML_2]\ S), (in(doc(t, b, H), v)\ C), U, n)\\ &\rightarrow((v[ML_1\ ;\ approve(t) \mid ML_2]\ S), (in(doc(t, b, (apv(v)\ H)), v)\ C), U, n)\end{aligned}$$

Definition 24 (*Approve-2*). The rule *approve-2* is:

$$\begin{aligned}&((v[ML_1 \mid approve(t, k)\ ;\ ML_2]\ S), (in(doc(t, b, (k\ H)), v)\ C), U, n)\\ &\rightarrow((v[ML_1\ ;\ approve(t, k) \mid ML_2]\ S), (in(doc(t, b, (apv(v)\ kH)), v)\ C), U, n)\end{aligned}$$

4.2 Attacker Model

We adopt a quite simple attacker model in which an attacker can forge a document if it is in an untrusted division. Each attacker belongs to the special kind of division called an attacker division.

Definition 25 (*Attacker division*). An *attacker division* is constructed by the function *attacker* : $Div \rightarrow Div$, which means that there is an attacker in the division.

Note that an attacker can only forge a document that has not yet been checked. We assume that our checking mechanism is completely trustworthy. In the real world, this means that we cannot forge other persons' signatures or seals. While this is a somewhat simplifying assumption, it is a good starting point since many risks can be identified even under this assumption. Future work will explore extending our attacker model to forging a signature similar to work done in cryptographic protocols.

An attacker uses a special message called attacker message.

Definition 26 (*Attacker message*). An *attacker message* is constructed by the following function:

$$forge: DocType \rightarrow SMsg$$

Definition 27 (*Forge*). The rule *forge* is:

$$\begin{aligned}&((attacker(v)\ [nil \mid forge(t);\ nil]\ S), (in(doc(t, true, h), v)\ C)\ (vU), s(n))\\ &\rightarrow(S, (in(doc(t, false, empty), v)\ C)\ (vU), n)\end{aligned}$$

This simple rule includes several important points. An attacker can forge a document: if that attacker belongs to an untrusted division, if the document is not a carbon copy, if the document is a real one and if the illegal activity bound is not zero. These conditions are represented by the left hand side of the rule. Also, note that if we apply this rule, and to a state of a BPF, in a backward search, the strand set of the resulting state will have a new element *attacker*(v)[nil | forge(t); nil]. This mechanism allows us to deal with an unbounded number of attackers by generating new instances as needed.

5. ANALYSIS OF MICRO VIEW MODEL

Here, we mainly focus on the micro view of a BPF and discuss how we can analyze risks involved in the BPF. Given a BPF we would like to know whether or not there exists any risk, and if so, we also want to know how the unwanted situation will occur. We have implemented our framework as a prototype tool using Maude (Clavel *et al*, 2007).

Thanks to analysis capabilities provided by rewriting logic, Document Logic supports several different levels of analysis. This is quite important because there is a trade-off between analysis power and computational cost according to the analysis method we choose. Our framework has three major levels of analysis such that:

- execution,
- reachability analysis using forward execution,
- reachability analysis using backward execution.

We can execute BPF to see what will happen. This is almost the same idea as testing a program by execution. In a concurrent process situation like BPF, this analysis gives us quite limited information, because we have lots of different execution paths.

The reachability analysis using forward execution is much more powerful than just executing a BPF. In this analysis method, a user gives an initial state of a BPF and a pattern (a term with variables) of final states. Then, Maude searches all the execution paths from the initial state and outputs all the final states that match the pattern. This search is done by rewriting with R_{DL}. Maude also allows the user to specify a temporal property in LTL (Linear Temporal Logic) instead of just giving a pattern of final states. In other words, we can model check a BPF by using the Maude LTL model checker. To know all the risks involved in a BPF, we should manually enumerate all the initial states and run the analysis for all of them.

The reachability analysis using backward execution is more powerful compared to the former two methods. A user gives a pattern of final states, and Maude searches for all the initial states from which the final states that match the pattern are reachable. Using this approach, the user does not have to specify in advance the number of attacker instances needed to reach the specified state. The following is an example of an illegal situation for Figure 1:

$$\begin{array}{l} ((client[create(order); snd(order, sales); rec(invoice, sales) \mid nil] \\ sales(rec[order, client); create(invoice, order); snd(invoice, client) \mid nil]) \\ (in(doc(order, b_3, empty), v_1)\ in(doc(invoice, false, empty), client)), (sales), 2) \end{array}$$

This pattern means that *client* and *sales* have completed their work, and the *invoice* is forged and *client* has it. However, we do not care about the *order*, where it is (v_1 is a division variable), and whether or not it is forged (b_3 is a Boolean variable).

This search is done by narrowing with the backward form of the rules $\hat{R}_{DL}$. Using Maude's built-in narrowing and meta-programming capabilities, we have built a prototype tool. The built-in narrowing command takes a term and gives us a result. We have to collect all the results and select the initial states from the results (since the results include states that are not initial). We implement this analysis tool by using Maude's meta-programming facility. We note that the prototype tool can be easily extended to work with either forward or backward rules, thus allowing both forward and backward analysis. Before we present details on the analysis in Section 7, we investigate the necessary extensions to support macro view model analysis.

6. EXTENDING DOCUMENT LOGIC TO ANALYZE MACRO VIEW MODEL

We extend Document Logic to fit the macro view model of a BPF that has several sessions. A division may participate in several BPF instances and each BPF instance is identified by a session identifier. For analyzing the macro view model, we use the forward rules for forward reachability analysis (see next section).

Definition 28 (*Session*). A *session* is constructed by the following function:

$$session : Div \times SessionId \rightarrow Session$$

where *SessionId* is a set of session identifiers.

The definition of *Doc* is extended to have information that identifies the session in which the document is created.

Definition 29 (*Document* (macro view version)). A *document* is constructed by the following function:

$$doc: DocType \times Bool \times SessionId \times EvidenceHistory \rightarrow Doc$$

A strand now has *Session* information instead of just having *Div*. The rules of Document Logic are also extended according to the above definitions in the obvious way. Exceptions are the rules for create and check that handle session identifiers as follows. When creating a document from another document, the new document inherits the session identifier, as can be seen in the new rules for *Create-2* and *Check-2*.

Definition 30 (*Create-2*). The rule *create-2* (macro view version) is:

$$\begin{aligned}&((e[ML_1 \mid create(t_1, t_2) \,;\, ML_2]S), (in(doc(t_2, true, i, H), e)\; C), U, n)\\ &\rightarrow((e[ML_1 \,;\, create(t_1, t_2) \mid ML_2]S), (in(doc(t_1, true, i, empty), e)\\ &\quad (in(doc(t_2, true, i, H), e)\; C), U, n)\end{aligned}$$

where e is a variable for *Session* and i is a variable for *SessionId*.

Definition 31 (*Check-2* (macro view version)). The rule *check-2* is:

$$\begin{aligned}&((e[ML_1 \mid check(t_1, t_2) \,;\, ML_2]S), (in(doc(t_1, b, i, H_1), v)\; in(doc(t_2, b, i, H_2), v)\; C), U, n)\\ &\rightarrow((e[ML_1 \,;\, check(t_1, t_2) \mid ML_2]S), (in(doc(t_1, b, i, (ch(t_2)\; H_1)), e)\\ &\quad (in(doc(t_2, b, i, H_2), e)\; C), U, n)\end{aligned}$$

where e is a variable for *Session* and i is a variable for *SessionId*.

The above definition of *check-2* means that document t_1 can pass the check with document t_2 if both documents have the same authenticity and the same session identifier. *SessionId* can be understood similar to a "serial number" that is used to bind several documents to a session. The other rules for *create* and *check* must be extended in a similar way.

The most interesting question is how we should extend the attacker rules. We propose an extension so that an attacker can exchange two documents of different sessions that have the same document type. The following is the rule for this new attack:

Definition 32 (*Exchange*). The rule *exchange* is:

$((attacker(v)[nil \mid exchange(t); nil]\ S), (in(doc(t, b_1, o_1, H_1), session(v, i_1))$
$(in(doc(t, b_2, o_2, H_2), session(v, i_2))\ C), (v\ U), n)$
$\rightarrow((attacker(v)[exchange(t) \mid nil]\ S), (in(doc(t, b_1, o_1, H_1), session(v, i_2))$
$(in(doc(t, b_2, o_2, H_2), session(v, i_1))\ C), (v\ U), s(n))$

An attacker can exchange documents if the attacker and the two documents are in the same untrusted division. Note that the authenticity of exchanged documents is not changed.

We do not claim that the above attacker rule constitutes a complete attacker model, but it is sufficient to demonstrate a rather intricate attack in our case study. Other attacker rules are possible. For example, an attacker may just intercept documents from one session and substitute them instead of other documents in another session, without changing anything in the session from which the document was intercepted, or an attacker can forge a document with arbitrary session identifier.

7. CASE STUDY

The example BPF of Figure 2 is a sequence diagram representing a simple wholesale BPF. We show the results of two analyses for this example: (1) backward reachability analysis of the micro view model, and (2) forward reachability analysis of the macro view model. These two analyses will highlight the differences between micro and macro view models.

7.1 Backward Reachability Analysis of Micro View Model

We consider the following final goal, which we have divided into four components: *StrandSet*, *Cabinet*, *UntrustedDivision*, and *IllegalActivityBound*:

StrandSet =
(*client*[*create-cc*(*order*); *snd*(*order, sales*); *rec*(*ack, sales*); *check*(*ack, order*);
rec(*invoice, shipping*); *check*(*invoice, order*); *create*(*receipt, invoice*); *snd*(*receipt, sales*) | *nil*]
sales[*rec*(*order, client*); *create*(*ack, order*); *snd*(*ack, client*); *create*(*request, order*);
snd(*request, shipping*); *approve*(*order*); *rec*(*report, shipping*); *rec*(*receipt, client*);
check(*receipt, order, apv*(*sales*)); *check*(*report, order, apv*(*sales*)) | *nil*]
shipping[*rec*(*request, sales*); *create-cc*(*invoice, request*); *snd*(*invoice, client*); *create*(*report, invoice*);
snd(*report, sales*); | *nil*])
Cabinet =
$(in(doc(invoice, b_1, H_1), v_1)\ in(cc(doc(order, b_2, H_2)), v_2)\ (in(doc(ack, b_3, H_3), v_3)$
$(in(doc(order, b_4, H_4), v_4)\ in(doc(report, false, (ch(order), H_5)), sales_2)\ (in(doc(receipt, b_6, H_6), v_6)$
$(in(cc(doc(invoice, b_7, H_7)), v_7)\ in(doc(request, b_8, H_8)), v_8))$
UntrustedDivision = (*sales shipping*)
IllegalActivityBound = 2

This query means “Is there a case for which the sales division finally has a forged report that is checked with *order*?” This is encoded in the above goal as part of the cabinet as follows:

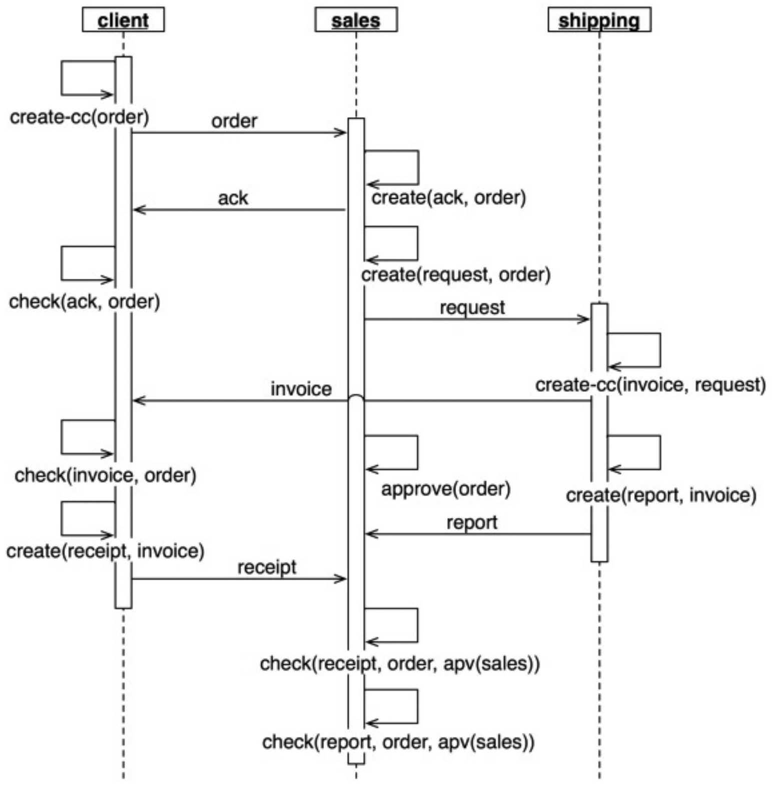

Figure 2: Whole sales

$in(doc(report, false, (ch(order), H_5)), sales)$

The above goal (i.e., the definition of *StrandSet*, *Cabinet*, *UntrustedDivision*, and *IllegalActivityBound*) is given as the pattern of the final state and our Maude tool searches for all initial states from which the final states that match the pattern are reachable. Executing this reachability analysis using backward execution (that is, reversing the arrows in all the forward rules specified in Section 4 so that they become backward rules) is triggered using the command rew solutions(term). Maude> is the Maude input prompt, and executing rew solutions(term) gives the following result:

```
Maude> rew solutions(term) .
rewrite in NARROW : solutions(term) .
rewrites: 98525 in 186597ms cpu (280944ms real)
                                    (528 rewrites/second)
result StateList: nil-sl
```

The resulting "StateList: nil-sl" means that there is no such initial state, and thus, there is no initial state from which a final state that matches the given goal pattern could be reached.

We can modify the goal pattern to analyze a different aspect of the BPF. For example, we modify the last message of the sales division in the strand set from *check*(*report*, *order*, *apv*(*sales*)) to *check*(*report*, *order*). This new goal pattern is one in which the sales division checks the report with

```
client[ nil-ml |
  create-cc(order) ;
  snd(order, sales) ;

        .
        .
        .

  create(report, invoice) ;
  snd( report, sales) ; nil-ml]
attacker(sales)[ nil-ml |
  forge(order) ; nil-ml]
attacker(sales)[ nil-ml |
  forge(report) ; nil-ml] empty
```

```
client[ nil-ml |
  create-cc(order) ;
  snd(order, sales) ;

        .
        .
        .

  create(report, invoice) ;
  snd(report, sales) ; nil-ml]
attacker(sales)[ nil-ml |
  forge(order) ; nil-ml]
attacker(shipping)[ nil-ml |
  forge(report) ; nil-ml] empty
```

Figure 3: Possible Risks

an order that has no evidence history. The old goal pattern required that the sales division checks the report with an order that was approved by the sales division.

If we execute the same query "rew solutions(term)" but with the new goal as the term to be re-written, we get the result shown in Figure 3. This result shows that there are two possibilities: to forge both the *order* and the *report* in sales, or to forge the *order* in sales and forge the *report* in shipping.

Formulating the final goal that will be used for backward reachability analysis can be done for a given BPF. Our example illustrates that the final goal must define all strand sets involved in the BPF, all documents exchanged in the BPF that end up in the cabinet and the divisions that are untrusted. Finally, a document in the cabinet that is forged but passed checks must be identified. The strand sets model the business process of interest, and will be the same for all the analyses to be performed on the model, and many of the cabinet items will be common to all the analyses. Thus the main work is defining the bad documents to be considered. The illegal activity bound can be set experimentally, slowly increasing the bound until an error is found or, if the analysis does not uncover risks, until the analyst considers the achieved assurance level as sufficient.

While our example illustrated the goal definition for a forged report that was checked with an order, a similar goal template would suffice to capture other risks as pointed out in Section 1.3, such as an office worker in the sales division forging an invoice or an order. For example, the goal state for a forged invoice would be as follows:

StrandSet =
(*client*[*create-cc*(*order*); *snd*(*order, sales*); *rec*(*ack, sales*); *check*(*ack, order*);
rec(*invoice, shipping*); *check*(*invoice, order*); *create*(*receipt, invoice*); *snd*(*receipt, sales*) | *nil*]
sales[*rec*(*order, client*); *create*(*ack, order*); *snd*(*ack, client*); *create*(*request, order*);
snd(*request, shipping*); *approve*(*order*); *rec*(*report, shipping*); *rec*(*receipt, client*);
check(*receipt, order, apv*(*sales*)); *check*(*report, order, apv*(*sales*)) | *nil*]
shipping[*rec*(*request, sales*); *create-cc*(*invoice, request*); *snd*(*invoice, client*); *create*(*report, invoice*);
snd(*report, sales*); | *nil*])
Cabinet =
$(in(doc(invoice, false, (ch(order)\ H_1)), client)\ in(cc(doc(order, b_2, H_2)), v_2)$
$in(doc(ack, b_3, H_3), v_3)\ (in(doc(order, b_4, H_4), v_4)\ in(doc(report, b_5, H_5)), v_5)$
$in(doc(receipt, b_6, H_6), v_6)\ in(cc(doc(invoice, b_7, H_7)), v_7)\ in(doc(request, b_8, H_8), v_8))$
UntrustedDivision = (*sales shipping*)
IllegalActivityBound = 2

Here *StrandSet*, *UntrustedDivision*, and *IllegalActivityBound* are the same as in the previous example. The cabinet part of the goal is obtained by changing the authenticity component of the first invoice document to false and making the authenticity component of the report document a variable.

7.2 Forward Reachability Analysis of Macro View Model

The macro view model deals with unbounded number of sessions. Thus, performing backward reachability analysis comes with a huge computational cost. Although *forward reachability analysis* is a somewhat adhoc analysis method, it proves to be quite powerful if one can find a proper initial state. The computational cost is much smaller than that of backward reachability analysis and we can adopt an iterative analysis approach in which we increase the analysis complexity by increasing the numbers of sessions until we find an error, or become convinced that more simultaneous sessions are sufficiently unlikely.

In our case study we fix the number of sessions to two sessions, i.e. $SessionId = \{i_1, i_2\}$. We have three divisions and each has two sessions, which means that we have six session instances: *session*(*client*, i_1), *session*(*client*, i_2), *session*(*sales*, i_1), *session*(*sales*, i_2), *session*(*shipping*, i_1), and *session*(*shipping*, i_2).

We assume four attackers as follows:

attacker(*sales*)[*nil* | *forge*(*order*); *nil*] *attacker*(*sales*)[*nil* | *exchange*(*order*); *nil*]
attacker(*sales*)[*nil* | *exchange*(*order*); *nil*] *attacker*(*shipping*)[*nil* | *forge*(*order*); *nil*]

The risk we want to analyze is the same as the previous analysis. The initial state is as follows:

StrandSet =
(*session*(*client, s*1)[*create-cc*(*order*); *snd*(*order, sales*); *rec*(*ack, sales*); *check*(*ack, order*);
rec(*invoice, shipping*); *check*(*invoice, order*); *create*(*receipt, invoice*); *snd*(*receipt, sales*) | *nil*]
(*session*(*client, s*2)[*create-cc*(*order*); *snd*(*order, sales*); *rec*(*ack, sales*); *check*(*ack, order*);
rec(*invoice, shipping*); *check*(*invoice, order*); *create*(*receipt, invoice*); *snd*(*receipt, sales*) | *nil*]
(*session*(*sales, s*1)[*rec*(*order, client*); *create*(*ack, order*); *snd*(*ack, client*); *create*(*request, order*);
snd(*request, shipping*); *approve*(*order*); *rec*(*report, shipping*); *rec*(*receipt, client*);
check(*receipt, order, apv*(*sales*)); *check*(*report, order, apv*(*sales*)) | *nil*]
(*session*(*sales, s*2)[…]
(*session*(*shipping, s*1)[…]
(*session*(*shipping, s*2)[…]
attacker(*sales*)[*nil* | *forge*(*order*); *nil*]
attacker(*sales*)[*nil* | *exchange*(*order*); *nil*]
attacker(*sales*)[*nil* | *exchange*(*order*); *nil*]
attacker(*shipping*)[*nil* | *forge*(*report*); *nil*]
Cabinet = ()
UntrustedDivision = (*sales shipping*)
IllegalActivityBound = 4

Figure 4 shows one of the results we got by performing forwards reachability analysis. The analysis reveals a quite subtle attack pattern. We call the *order* that is created in session #1 *order*$_1$ and the *order* that is created in session #2 *order*$_2$. An attacker forged *order*$_1$ right after the *receipt* is checked. By using this forged *order*$_1$, report cannot pass the succeeding check, because *order*$_1$ does not have the evidence of approval. So, attacker intercepts *order*$_1$ and *order*$_2$ that is waiting for approval in session #2, and exchanges them. In session #2, *order*$_1$ can be approved by a manager of

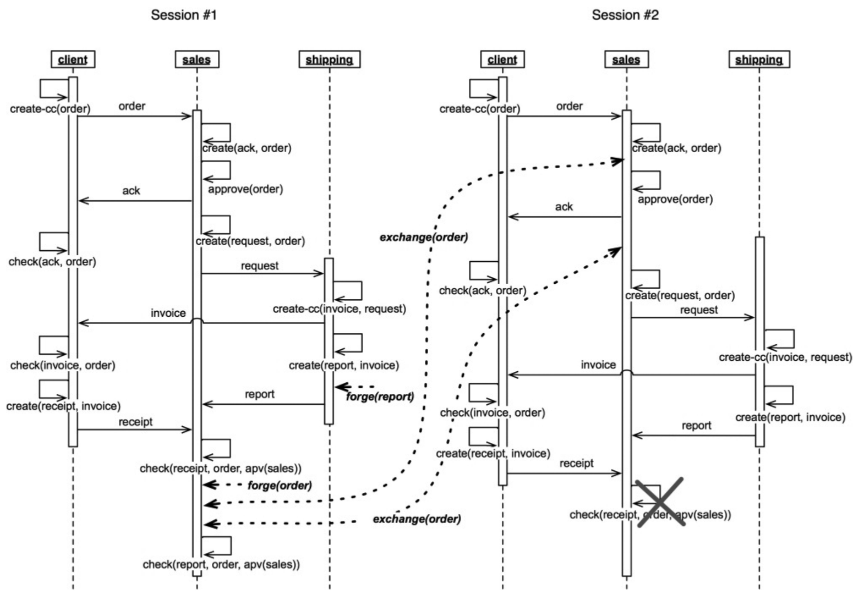

Figure 4: A risk in macro view model.

the sales division because the operation *approve* does not depend on the session identifier. The reason is that we assume that in a real business process the operation approve is performed by a manager to ensure the content of the document and to confirm conditions, such as credit limit, regulations, and standards, all of which are independent of a specific session. Right after $order_2$ gets approval, an attacker exchanges $order_1$ and $order_2$ again. The forged *report* is then checked with the forged $order_1$ which has the evidence of approval and, thus, passes the check. This attack cannot be achieved by using only one session.

Of course, in session #2, the last check fails and the attack would be discovered. However, by this time, the attack could have already been successfully completed and caused real harm to the business. In addition, even though the attack might have been discovered, it might still take some time for a human to trace the origin of the forged documents and determine the exact attack pattern. From this simple example, it becomes clear that more sophisticated attacks (e.g., including more sessions), would be even harder to find for a human, yet can be detected by means of automated analysis.

8. CONCLUSION

Our research goal is to model social systems that involve many kinds of risks. Social systems, like business processes, contain many complex concepts, such as illegality, profits, and trust. These concepts usually do not appear in the engineering systems that are the usual subject of formal analysis, although they have some similarities with cryptographic protocols. We want to build formal models that can explain such concepts well and use them to help experts in the design of safe business processes. Document Logic is an example of this attempt.

Document Logic is a simple yet powerful framework to infer risks in business process flows. We focus on flows of documents and build a set of inference rules based on document authenticity and a simple trust model. Document Logic can be seen as a specialized logic for risk analysis of business process flow. From this point of view, rewriting logic together with its Maude implementation plays a crucial role. By using rewriting logic, we can easily implement our experimental concepts by means of rewrite rules. This is useful in understanding business activities more precisely, such as what does checking a document mean? When can an attacker forge a document? By using Maude, this specialized logic can immediately become a prototype tool that automatically answers our queries. Also, the meta- level programming facility of Maude allows us to build such a special function quite readily.

However, there is much to do to achieve our goal. The current version of Document Logic cannot talk about individuals in a division and their authorities. Authority is quite an important aspect when we consider risks in a BPF. Adding authority to the rules of Document Logic is not so difficult; however, the number of states we have to deal with can easily explode. Thus, we will need to find heuristic techniques to cut down the search spaces. We hope that this can be done by studying real business knowledge.

Also, we made the assumption of a trustworthy checking mechanism in the attacker model. This is similar in spirit to the perfect encryption option made in many cryptographic protocol analysis formalisms. Even with such strong assumptions, many problems can be found. Our approach has been to start with a very simple logic and attack model, develop analysis methods, and understand the types of risks that can be found. A more extensive attack model is an important topic for future work.

ACKNOWLEDGMENT

We thank José Meseguer and Kokichi Futatsugi for useful suggestions and comments on this work. This research has been partially supported by a Senshu University Soma Research Fund grant.

REFERENCES

BARTH, A., DATTA, A., MITCHELL, J.C. and SUNDARAM, S. (2007): Privacy and utility in business processes. In Proceedings of 20th IEEE Computer Security Foundations Symposium, 279–294. *IEEE Computer Society*.

CLAVEL, M., DURÁN, F., EKER, S., LINCOLN, P., MARTÍ-OLIET, N., MESEGUER, J. and TALCOTT, C. (2007): All about Maude – A high-performance logical framework, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag.

DENKER, G., MESEGUER, J. and TALCOTT, C.L. (1998): Protocol specification and analysis in Maude. In HEINTZE, N. and WING, J. editors, *Proceedings of Workshop on Formal Methods and Security Protocols*, June 25, 1998, Indianapolis, Indiana.

DOLEV, D. and YAO, A. (1983): On the security of public key protocols. *IEEE Trans. on Information Theory*, 29 (2):198–208.

ESCOBAR, S., MEADOWS, C. and MESEGUER, J. (2006): A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. *Theoretical Computer Science*, 367:162–202.

OBJECT MANAGEMENT GROUP (2009): Business Process Modeling Notation (BPMN) ver. 1.2. http://www.omg.org/spec/BPMN/1.2/, 2009.

MESEGUER, J. (1992): Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155.

MESEGUER, J. (1998): Membership algebra as a logical framework for equational specification. In PARISI PRESICCE, F. editor, Recent trends in algebraic development techniques. *Proc. 12th International Workshop*, volume 1376 of *Lecture Notes in Computer Science*, 18–61. Springer-Verlag.

OASIS Web Services Business Process Execution Language (WSBPEL) TC. (2007): Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

FÁBREGA, F.J.T., HERZOG, J. and GUTTMAN, J.D. (1998): Strand spaces: Why is a security protocol correct? In 1998 IEEE Symposium on Security and Privacy. *IEEE Computer Society Press*, May.

VAN DER AALST, W. and VAN DE GRAAF, W. (2002): Workflow systems. In GIRAULT, C. and VALK, R. editors, *Petri Nets for System Engineering*, 1:507–540. Springer-Verlag.

BIOGRAPHICAL NOTES

Dr Shusaku Iida is a professor at the School of Network and Information at Senshu University in Kawasaki, Kanagawa, Japan. He received his PhD in information science from the Japan Advanced Institute of Science and Technology in 1999. His research interests include algebraic specification languages, formal models of social systems, and business process modeling.

Shusaku Iida

Dr Grit Denker is a senior computer scientist in the Computer Science Laboratory of SRI International. Prior to joining SRI, she was an assistant professor at the Technical University of Braunschweig in Germany. She received her PhD in computer science from the Technical University of Braunschweig in 1995. Dr Denker's expertise includes semantic models and reasoning, specification and verification of distributed systems, semantic web and semantic web services, and policy languages.

Grit Denker

Dr Carolyn Talcott is a program manager in the Computer Science Laboratory of SRI International, where she leads the Symbolic Systems Technology group. Dr Talcott holds a PhD in computer science from Stanford University and a PhD in chemistry from the University of California, Berkeley. She has over 20 years experience in formal modeling and analysis. At SRI Dr Talcott is leading research in symbolic systems biology, security protocol analysis, and formal analysis applied to networked cyber physical systems.

Carolyn Talcott