HAL
open science

Decentralized Enforcement of Artifact Lifecycles

Sylvain Hallé, Raphaël Khoury, Antoine El-Hokayem, Yliès Falcone

To cite this version:

Sylvain Hallé, Raphaël Khoury, Antoine El-Hokayem, Yliès Falcone. Decentralized Enforcement of Artifact Lifecycles. EDOC 2016, Sep 2016, Vienne, Austria. hal-01365315

HAL Id: hal-01365315

https://inria.hal.science/hal-01365315

Submitted on 13 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Decentralized Enforcement of Artifact Lifecycles

Sylvain Hallé, Raphaël Khoury

Laboratoire d'informatique formelle
Université du Québec à Chicoutimi, Canada
Antoine El-Hokayem, Yliès Falcone

Université Grenoble Alpes, Inria, LIG, Grenoble, France

***Abstract*—Artifact-centric workflows describe possible executions of a business process through constraints expressed from the point of view of the documents exchanged between principals. A sequence of manipulations is deemed valid as long as every document in the workflow follows its prescribed lifecycle at all steps of the process. So far, establishing that a given workflow complies with artifact lifecycles has mostly been done through static verification, or by assuming a centralized access to all artifacts where these constraints can be monitored and enforced. We present in this paper an alternate method of enforcing document lifecycles that requires neither static verification nor single-point access. Rather, the document itself is designed to carry fragments of its history, protected from tampering using hashing and public-key encryption. Any principal involved in the process can verify at any time that a document's history complies with a given lifecycle. Moreover, the proposed system also enforces access permissions: not all actions are visible to all principals, and one can only modify and verify what one is allowed to observe.**

I. Introduction

The execution of a business process is often materialized by the successive manipulation of a document passing from one agent to the next. However, the document may have constraints on the way it is modified, and by whom: we call this the lifecycle of a document. In the past decade, *artifact-centric* business processes have been suggested as a modelling paradigm where business process workflows are expressed solely in terms of document constraints: a sequence of manipulations is deemed valid as long as every document (or "artifact") in the workflow follows its own prescribed lifecycle at all steps of the process. In this context, an artifact becomes a stateful object, with a finite-state machine-like expression of its possible modifications. As we shall see in Section II, this paradigm can be applied to a variety of situations, ranging from medical document processing to accounting and even electronic-pass systems such as smart cards.

Central to the question of business processes execution is the concept of *compliance checking*, or the verification, through various means, that a given implementation of a business process satisfies the constraints associated with it. Transposed to artifact-centric business processes, this entails one must provide some guarantee that the lifecycle of each artifact involved is respected at all times.

There are currently two main approaches to the enforcement of this lifecycle, which will be detailed in Section III. A first possibility is that all the peers involved in the manipulations trust each other and assume they perform only valid manipulations of the document; this trust can be assessed through testing or static verification of the peer's implementation. Otherwise, all peers can trust a third party, through which all accesses to the document need to be done; this third-party is responsible for enforcing the document's lifecycle, and must prevent invalid modifications from taking place. The reader shall note that both scenarios require some form of external trust, which becomes an entry point for attacks. In the first scenario, a single malicious user can thwart the enforcement of the lifecycle and invalidate any guarantees the other peers can have with respect to it. In the second scenario, reliance on a third party opens the way to classical mistrust-based attacks, such as man-in-the-middle.

In this paper, we present a mechanism for the distributed enforcement of a document's lifecycle, in which every peer can individually check that the lifecycle of a document it is being passed is correctly followed. This system, presented in Section IV, requires neither centralized access to the document, nor trust in other peers that are allowed to manipulate it. Rather, the document itself is designed to carry fragments of its history, called a *peer-action sequence*. This sequence is protected from tampering through careful use of hashing and public-key encryption. Using this system, any peer involved in the business process can verify at any time that a document's history complies with a given lifecycle, expressed as a finite-state automaton. Moreover, the proposed system also enforces access permissions: not all actions are visible to all principals, and one can only modify and verify what one is allowed to observe.

To illustrate the concept, Section V describes an implementation of these principles in a simple command-line tool that manipulates dynamic PDF forms. Peer-action sequences are injected through a hidden field into a PDF file, and updated every time the form is modified through the tool. As a result, it is possible to retrieve the document's modification history at any moment, verify its authenticity using a public keyring, and check that it complies with a given policy.

The section concludes with a few discussion points. In particular, it highlights the fact that, using peer-action sequences, the compliance of a document with a given lifecycle specification can easily be checked. Taken to the extreme, lifecycle policies can even be verified without resorting to any workflow management system at all: as long as documents are properly stamped by every peer participating in the workflow, the precise way they are exchanged (e-mail, file copying, etc.) becomes irrelevant. This presents the potential of greatly simplifying the implementation of artifact-centric workflows, by dropping many assumptions that must be fulfilled by current systems.

II. Document Lifecycles

We shall now describe a number of distinct scenarios, taken from past literature, that can be modelled as sets of constraints over the lifecycle of some document. In the following, the term *document* will encompass any physical or logical entity carrying data and being passed on to undergo modifications. This can represent either a physical memory card, a paper or electronic form, or more generally, any object commonly labelled as an "artifact" in some circles.

A special case of "lifecycle" is one where conditions apply on snapshots of documents taken individually, irrespective of their relation with previous of subsequent versions of this document. For example, the lifecycle could simply express conditions on what values various elements of a document can take, and be likened to integrity constraints. However, in the following, we are more interested in lifecycles that also involve the *sequence* of states in which the document is allowed to move through, and the identity of the effectors of each modification.

A. Medical Document Processing

As a first example, we consider a medical health care process, introduced by Bielova et al. [7] and illustrated in Figure 1. Medical data is inherently sensitive and the inappropriate manipulations of medical document can have far-reaching implications. As a consequence, medical processes are subject to rigid controls, documented in standards. Figure 1 models one such process, namely medication dispensation. The information related to medication dispensation is recorded in a file F, which allows doctors to track the medications of their patients, drug-trial administrators to monitor the occurrence of side-effects and hospitals to get reimbursements from the states for the drugs they administer.

As described in [6], the expected sequence of manipulation proceeds as follows:

1) First, a doctor selects a drug.
2) Depending on the drug, the doctor may be obligated to review therapeutic notes before proceeding further.
3) If the drug is part of an ongoing trial, the doctor must record the trial protocol in the prescription.
4) The doctor inputs any other relevant detail in the prescription.
5) The availability of the drug in the stock is checked.
6) If the drug is unavailable in the stock, its availability in the ward is checked.
7) If the drug was available, either in the stock or in the ward, the process is complete, otherwise, the doctor must select an alternative drug.

This process was designed to ensure compliance with the standards and regulation in place in the Italian region of Lombardia [6].

B. Accounting Processes

Another context in which the sequencing of document manipulations is particularly sensitive is banking. In this case, restriction on the workflow of document manipulations ensures that the proper banking laws and regulations as well as with the proper precautions that ensure the prudent management of money.

Rao *et al.* recently [26] studied banking document workflow and proposed a novel formalism for stating the restrictions governing document workflows. Their formalism, the Process Matrix, is strictly more expressive than BPMN as it allows users to place conditional restrictions on the obligation to perform certain steps.

Figure 2 shows the running example used in [26], a loan application process. Each row of the figure represents an activity of the process, listed in the first column. The next three column indicate the access rights for each of the three roles (applicant, case worker and manager) that a principal can possess in this process. For instance, the applicant can write-out an application, which can then be read by both the case worker and the manager, but only the manager can apply the second approval to a demand for a loan. The next column lists the constraints on the sequencing between activities. It distinguishes between regular predecessor, with their usual meaning, and logical predecessor, indicated with an asterisk (*). If activity A is a logical predecessor to activity B, then anytime activity A is re-executed, activity B must also be re-executed. The final column describes optional Boolean activity conditions that may render an an activity superfluous. In our example, the second approval can be omitted if the predicate *Rich* holds.

C. Data Integrity Policies

The scheme under consideration could also be useful in regards to the enforcement of several classes of Data Integrity policies. All of them can be stated as finite automata [15].

Assured pipelines [8] facilitate the secure transfer of sensitive information over trust boundaries by specifying which data transformation must occur before any other data processing. For instance, assured pipelines can be specified to ensure that confidential data is anonymized before being publicly disseminated, or that user inputs be formatted before being inputted into a system.

A Chinese Wall policy [9] can put in place to prevent conflicts of interest from occurring. For instance, a Chinese wall policy can be set up to prevent a consultant from advising two competing firms, or an investor from suggesting placements in a company in which he holds interest.

«««< HEAD Sobel *et al.* propose a trace-based enforcement model of the Chinese wall policy, enriched with useful notions of data-relinquishing and time-frames, for which the data management scheme proposed in this paper is suited [30]. In their framework, each object o is associated with a list of action-principal pairs, sequentially listing the actions (either *create* or *read*) each principal performed on the object. On a well-formed object, the list begins with a single *create* event, followed by a series of *reads*. The policy is stated as a set of conflicts of interests $\mathscr{C} \in \mathbb{P}(O)$. Each object O_i is associated with its conflict of interest $\mathscr{C}_i$, that lists the other objects that conflicts with it. The enforcement of the Chinese wall policy is ensured by preventing any user who has accessed a object in set $\mathscr{C}_i$ from accessing object O_i.

Finally, the low-water-mark policy was designed by Biba [5] to capture the constraints that ensure data integrity. In this model, each subject, and each data object, is mapped to a integrity level indicating its *trustworthiness*. A subject can only

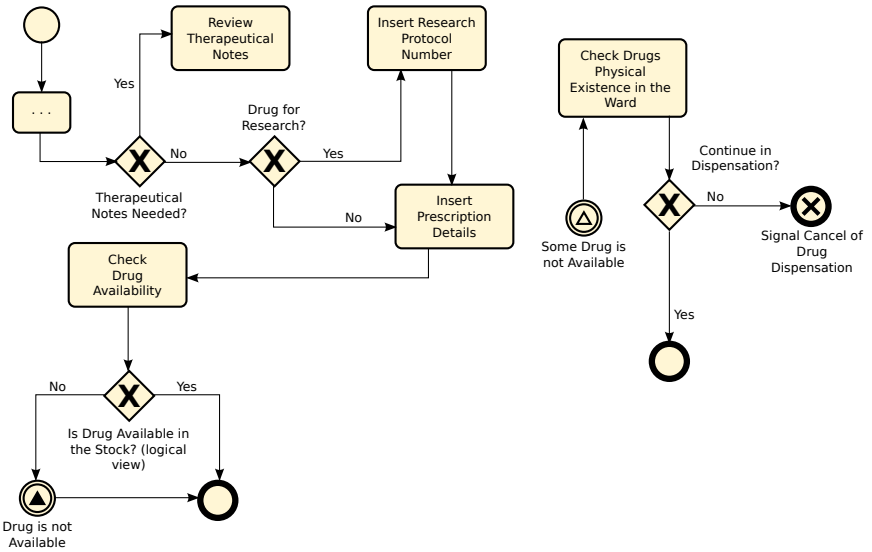


Figure 1: BPMN model drug dispensation process, from Bielova *et al.* [6]

	Activities	Roles			Prede-cessors	Activity Condition
		App	CW	Mgr		
1	Application	W	R	R		
2	Register Customer Info	W	W	W		
3	Approval 1	W	W	R	$\overset{*}{1}$,2	
4	Approval 2	D	R	W	$\overset{*}{1}$,2	¬Rich
5	Payment	R	W	R	$\overset{*}{3}$,$\overset{*}{4}$	¬Hurry∧ Accept
6	Express Payment	R	W	R	$\overset{*}{3}$,$\overset{*}{4}$	Hurry∧ Accept
7	Rejection	R	W	R	$\overset{*}{3}$,$\overset{*}{4}$	¬Accept
8	Archive	D	W	R	$\overset{*}{5}$,$\overset{*}{6}$,$\overset{*}{7}$	

Figure 2: Process Matrix for a loan application, from [26]

write to objects that are equal or below its integrity level, and can only read object that are higher or equal of its own integrity level. This prevents subjects and objects from being tainted with unreliable (low-integrity) data.

D. Other Examples

Smart cards, such as MIFARE Classic[1], are used to grant access to public transit. They record the number of access tokens their carriers currently holds, as well as a trace of his previous journeys in the system. The card is edited by a *card reader*.

In this case, the source of mistrust is not the readers, but the carrier of the card. For example, one does not wish the same card to enter twice from the same station, which would likely indicate an attempt at using the same card to get two people in. This is an example of a lifecycle property of the card. The information contained in the MIFARE card could also be used to allow or disallow transfers from one public transit route to another, with any applicable restriction captured in the lifecycle policy.

Similar cards are used in several contexts including library cards, hotel key cards, membership cards, Social welfare, car rentals and access to amusement parks or museums.

III. Enforcing Document Lifecycles

In the Business Process community, constraints on document lifecycles have been studied in the context of "object behaviour models". The most prominent form of such model is artifact-centric business process modelling [4], [22], [23], [28], [31]. In this context, various documents (called "artifacts") can be passed from one peer to the next and be manipulated. Rather than (or in addition to) expressing constraints on how each peer can execute, the business process is defined in terms of the lifecycle of the artifacts involved: any sequence of manipulations that complies with the lifecycle of each artifact is a valid execution of the process.

The specification of document lifecycles can be done in various ways. For example, the Business Entity Definition Language (BEDL) [27] allows the specification of lifecycles to business entities as finite state machines. Another possible way of modelling the lifecycle of these artifacts is the Guard-Stage-Milestone (GSM) paradigm [22], which identifies four key elements: an information model for artifacts; milestones which correspond to business-relevant operational objectives; stages, which correspond to clusters of activity intended to achieve milestones; and finally guards, which control when stages are activated. Both milestones and guards are controlled in a declarative manner, based on triggering events and/or conditions. Other approaches include BPMN with data [25] and PHILharmonic flows [24].

While the specification of artifact lifecycles is relatively well understood, the question of *enforcing* a lifecycle specified in some way has been the subject of many works, which can be categorized as follows.

[1] http://www.mifare.net

A. Centralized Workflow Approaches

Many works on that topic rely on the fact that the artifacts will be manipulated through a workflow engine. Therefore, the functionalities required to enforce lifecycle constraints can be implemented directly at this central location, since all read/write accesses to the documents must be done through the system. This is the case, for example, of work done by Zhao *et al.* [32]. Similar work has been done on the database front: Atullah and Tompa propose a technique to convert business policies expressed as finite-state machines into database triggers [2]. Their work is based on a model of a business process where any modification to a business object ultimately amounts to one or many transactions executed on a (central) database; constraints on the lifecycle of these objects can hence be enforced as carefully written INSERT or UPDATE database triggers.

In contrast, the work we present in this paper does not require any centralized access to the artifacts being manipulated.

B. Static Verification

In other cases, knowledge of the workflow makes it possible to statically analyze it and make sure that all declarative lifecycle constraints are respected at all times [4], [10], [16], [20], [21], [32]. For example, Gonzalez *et al.* propose a way of symbolically representing GSM-based business artifacts, in such a way that model checking can be done on the resulting model [17].

However, verification is in general a much harder problem than preventing invalid behaviours from occurring at runtime; therefore, severe restrictions must imposed on the properties that can be expressed, or the underlying complexity of the execution environment, in order to ensure the problem is tractable (or even decidable). For example, [10] considers an artifact model with arithmetic operations, no database, and runs of bounded length. [4], [16] impose that domains of data elements be bounded, or that pre- and post-conditions refer only to the artifacts, and not their variable values [16]. As a matter of fact, just determining when the verification problem is decidable has become a research topic in its own right. For example, Calvanese *et al.* identify sufficient conditions under which a UML-based methodology for modelling artifact-centric business processes can be verified [11].

Furthermore, in a setting where verification is employed, one must *trust* that each peer involved in the process has been statically verified, and also that the running process is indeed the one that was verified in the first place. This hypothesis in itself can prove hard to fulfill in practice, especially in the case of business processes spanning multiple organizations. In contrast, the proposed work eschews any trust assumptions by allowing any peer manipulating an artifact to verify by itself that any lifecycle constraint has indeed been followed by everyone. Moreover, since lifecycle violations are checked at the time of execution (a simpler problem than static verification), our approach can potentially use very rich behaviour specification languages.

C. Decentralized Workflow Approaches

The correctness of the sequence of operations can also be checked at runtime, as the operations are being executed; this was attempted by one of the authors in past work [19]. This concept has also been suggested, e.g. for the enforcement of lifecycle constraints on RFID tags passing from one reader to the next [29].

It is also possible to reuse notions found in Decentralized Runtime Monitoring [3], [12]. Runtime monitoring consists in checking whether a run of a given system verifies the formal specification of the system. In this case, the lifecycle is the specification, and the sequence of modifications to the document is the trace to be verified. Decentralized runtime monitoring is designed with the goal to monitor decentralized systems, it is therefore possible to monitor decentralized changes to a document. The approach performs monitoring by progressing LTL —that is, starting with the LTL specification, the monitor rewrites the formula to account for the new modifications. However, at the cost of offering full decentralization, LTL progression could increase the size of the formula significantly as the sequence of actions grows. The growth rate poses a challenge to store the new formula in the document when storage space is small and sequence lengths are large. It is however possible to reduce the overhead significantly by using an automata-based approach [13], at the cost of communicating more between the various components in the decentralized system. This approach could be suitable for a specific type of lifecycles where interaction is frequent between the various parties.

In a similar way, in *cooperative runtime monitoring* (CRM) [18], a recipient "delegates" its monitoring task to the sender, which is required to provide evidence that the message it sends complies with the contract. In turn, this evidence can be quickly checked by the recipient, which is then guaranteed of the sender's compliance to the contract without doing the monitoring computation by itself. This differs from the approach presented in this paper in many respects. First, cooperative runtime monitoring expects the properties to be known in advance, and to belong to the NP complexity class; our proposed approach is independent from the lifecycle specification. Second, and most importantly, CRM does not protect the tokens exchanged between a client and a server; a request can be replaced by another, through a man-in-the-middle attack, and be accepted by the server so long as it is a valid continuation of the current message exchange. Finally, the approach is restricted to a single two-point, one-way communication link.

D. Cryptographic Approaches

Finally, some related works can also be found based on security and cryptography. In this context, work on "lifecycle" enforcement has mostly focused on preventing the mediator of the document (for example, the owner of a metro card) from tampering with its contents. Therefore, a common approach is to encrypt the document's content, using an encryption scheme where keys are shared between peers but are unknown to the mediator. This approach works in a context where peers do not trust the mediator, but do trust each other. Therefore, compromising a single peer (for example, by stealing its key) can compromise the whole exchange. In contrast, our proposed technique provides tighter containment in case one of the peers is compromised. For example, stealing the private key of one of the peers cannot be exploited to force violations of the lifecycle,

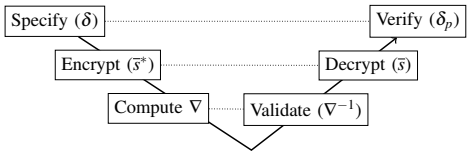


Figure 3: Lifecycle Enforcement

if the remaining peers still check for lifecycle violations and deny further processing to a document that contains one. As a matter of fact, we have seen how the peer-action sequence, secured by its digest, can in such a case be used to identify the peer responsible for this deviation of the lifecycle.

IV. Lifecycle Enforcement With Peer-Action Sequences

To alleviate the issues mentioned above, we describe in this section an original technique for storing a history of modifications directly into a document. Given guarantees on the authenticity of this history (which will be provided through the use of hashing and encryption), this technique allows any peer to retrieve a document, check its history and verify that it follows a lifecycle specification at any time.

In the following, we assume the existence of public key encryption/decryption functions; the notation $E[M,K]$ designates the result of encrypting message M with key K, while $D[M,K]$ corresponds to decryption. We also suppose the existence of a hash function $\hbar$, and assume to simplify notation that its set of output values is $\mathbb{H}$. We fix P to be the set of *peers*; each peer $p \in P$ possesses a pair of public/private encryption keys noted $K_{p,u}$, $K_{p,v}$, respectively. Peers belong to one or more access groups. The set of groups is G and consists of labels identifying each group. Groups are akin to the notion of *role* in classical access-control models such as RBAC [14]: we shall see that belonging to a group gives read/write access to a number of fields of the document under consideration.

Figure 3 illustrates our general approach to enforcing lifecycles. First we begin by defining the lifecycle δ, then show how a sequence can be encrypted to hide information from various groups, and how its digest is computed to ensure its integrity. Moreover, we explain how the sequence can be verified given its digest, then decrypted and verified by every peer p based on their permissions (δ_p).

A. Document Lifecycles

Let D be a set of *documents*, A be a set of *actions*; each action $a \in A$ is associated with a function $f_a : D \to D$ taking a document as an input, and returning another document as its output. A special document, noted $d_\emptyset$, will be called the *empty document*. Although not necessarily "empty", it represents the initial state in which all documents start prior to being modified by any peer. For the sake of simplicity, we assume that $d_\emptyset$ is unique in D.

A peer-action is a 4-tuple $\langle a,p,g,h\rangle \in (A \times P \times G \times \mathbb{H})$ consisting of an action a, identifying the peer responsible for this action p and identifying on behalf of which group g was the action taken (the purpose of the hash h will be explained later). We construct a sequence of peer-actions and denote it by $\overline{s}$. The set S contains all possible peer-action sequences, for given sets P, A, G and $\mathbb{H}$.

A document *lifecycle* specifies what actions peers are allowed to make on a document and in which order. It is represented by a function $\delta : S \to \{\top,\bot\}$. Intuitively, the function δ takes as input a peer-action sequence, and decides whether this sequence is valid ($\top$) or not ($\bot$).

As an example, let us consider the simple case of a metro card. In this case, P is the set of all metro stations and buses. $D = \{d_\emptyset\}$: the card does not carry any information, apart from metadata related to its history. The set of actions A is made of two actions $a_\downarrow$, and $a_\uparrow$, representing a user going in and out, respectively. We only define one group $G = \{pub\}$. The lifecycle function δ can then be defined as $\delta(\langle(p_1,a_1,g_1,h_1),\ldots,(p_n,a_n,g_n,h_n)\rangle) = \bot$ if and only if there exists an i such that $p_i = p_{i+1}$ and $a_i = a_{i+1} = a_\downarrow$. This corresponds to the case where two successive "in" actions occur at the same station.

We decentralize the specification by incorporating different groups. For each group $g \in G$ we consider a symmetric key S_g, we associate the *group lifecycle* function δ_g and we assign peers to groups using the predicate $M(p,g)$ indicating the peer p belongs to group g. δ_g specifies the actions allowed for a member of the group to make on the document. Therefore, a peer $p \in P$ belongs to the set of groups $G_p = \{g \mid M(p,g) = \top\}$. The lifecycle that p will verify is $\delta_p(s) : S \to \{\top,\bot\}$, with $\delta_p(s) = \bigwedge_{g\in G_p}(\delta_g(s))$, where $\top$ and $\bot$ are interpreted as Boolean `true` and `false` respectively. The lifecycle δ_p ensures that p can only verify the lifecycles of groups they belong to. We add the restriction that when a peer executes an action on a document, they execute it on behalf of one group only. We note that in this case, that a group lifecycle δ_g acts on the entire sequence. Therefore, the specification must be written in a way that δ_g is only concerned with the actions relevant to the group, ignoring the rest of the sequence and handling synchronization.

B. Encrypting a Sequence

Before storing the peer-action sequence in the document, we ensure confidentiality for group actions. For the scope of this paper, we seek to disallow non-group members to see which exact action has been taken, but not the fact that an action has been taken. A peer action $\langle a,p,g,h\rangle$ where peer p has taken an action a on behalf of g is encrypted as $\langle E[a,S_g],p,g,h\rangle$. The actual peer-action sequence stored in the document is $\overline{s}^* : (P \times \mathbb{H} \times G \times \mathbb{H})^*$. This ensures that members outside the group can see that the peer p has taken an action on behalf of the group g (thus are able to check $M(p,g)$), but cannot see which action (a) has been taken. Therefore they cannot know which f_a has been applied to the document.

C. Computing a Digest

The enforcement of a lifecycle is done by calculating and manipulating an history *digest*.

Definition 1 (Digest): Let $\overline{s}^* = (pa_1^*,\ldots pa_n^*)$ be an encrypted peer-action sequence of length n, and let $\overline{s'}^* = (pa_1^*,\ldots,pa_{n-1}^*)$ be the same sequence, trimmed of its last

peer-action pair, where $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ for $i \in [0,n]$. The digest of $\overline{s}^*$, noted $\nabla(\overline{s}^*)$, is defined as follows:

$$\nabla(\overline{s}^*) \triangleq \begin{cases} 0 & \text{if } n = 0 \\ E[\hbar(\nabla(\overline{s'}^*) \cdot a_n^* \cdot g_n), K_{v,p_n}] & \text{otherwise} \end{cases}$$

In other words, to compute the n-th digest of a given encrypted sequence $\overline{s}$, the peer p_n responsible for the last action a_n on behalf of the group g takes the last computed digest, encrypts a_n with the group key S_g, concatenates $E[a_n, S_g] \cdot g$, computes its hash, and encrypts the resulting string using its private key K_{v,p_n}. The use of the hash function ensures that the content to be encrypted is of constant length, and does not expand as new actions are appended to the document's history. Signing with the group id appended to the action is used to ensure the integrity of the group advertised. We note that when adding a new element to the encrypted sequence $\overline{s'}^*$ its hash will be $\nabla(\overline{s}^*)$.

The digest depends on the complete history of the document from its initial state. Moreover, each step of this history is encrypted with the private key of the peer having done the last action. Note that encrypting each tuple of the history separately would not be sufficient. Any peer could easily delete any peer-action pair from the history, and pretend some action did not exist. In the same way, a peer could substitute any element of the sequence by any other picked from the same sequence, in a special form of "replay" attack. Adding the action's position number in the digest would not help either, as any suffix of the sequence could still be deleted by anyone. Moreover, in this scheme, forging a new digest requires knowledge of other peers' private keys.

D. Checking a Digest

In addition to its data, a document should also carry the encrypted peer-action sequence and a corresponding digest. Checking that the sequence corresponds to the digest is done by verifying group membership and the hashes over the entire sequence.

Definition 2 (Verify Digest): Given an encrypted peer-action sequence $\overline{s}^* = (pa_1^*, \ldots, pa_n^*)$ of length n, and a digest d. Let $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ be the same sequence, trimmed of its last peer-action pair, and $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ for $i \in [0,n]$. The sequence $\overline{s}^*$ verifies d if and only if $\nabla^{-1}(\overline{s}^*, d)$ where:

$$\nabla^{-1}(\overline{s}^*, d) \triangleq \begin{cases} M(p_n, g_n) \wedge \exists \langle h, a^*, g \rangle : & \text{if } n > 0 \\ \quad D[d, K_{u,p_n}] = \hbar(h \cdot a^* \cdot g) & \\ \quad \wedge a^* = a_n^* \wedge g = g_n & \\ \quad \wedge h_n = \hbar(h \cdot a^* \cdot g) & \\ \quad \wedge \nabla^{-1}(\overline{s'}^*, h) & \\ \top & \text{otherwise} \end{cases}$$

Detecting a fraudulent manipulation of the digest or the peer-action sequence can be done in the following ways:

1) computing $D[d, K_{u,p_n}]$ produces a nonsensical result, indicating that the private key used to compute that partial digest is different from the one advertised by the peer-action sequence (in this case the tuple $\langle h, a^*, g \rangle$ cannot be generated);
2) computing $D[d, K_{u,p_n}]$ produces a string $h \cdot a^* \cdot g$ such that the action a^* and group g extracted from the digest does not match respectively a_n^* and g_n, contained in the sequence for that position
3) observing tampering with the hash $h_n \neq \hbar(h \cdot a^* \cdot g)$;
4) observing tampering with the groups ($g \neq g_n$) and observing that p_n is not in g_n.

We note that even if the action is hidden, it is still possible for the peer to verify that, at the very least, the peer p_n belongs to g_n and knows that p_n has taken an action.

E. Decrypting a Sequence

Once a peer validates the authenticity of a sequence, the peer will then have to decrypt the sequence to process the actions. The decryption of an encrypted peer-action sequence will depend on what the peer can see. The new sequence will depend on the groups the peer belong to.

Definition 3 (Decrypting a Sequence): Given an encrypted peer-action sequence $\overline{s}^* = (pa_1^*, \ldots, pa_n^*)$ of length n, and a peer $p \in P$. Let $\overline{s'}^* = (pa_1^*, \ldots, pa_{n-1}^*)$ be the same sequence, trimmed of its last peer-action pair, and $pa_i^* = \langle a_i^*, p_i, g_i, h_i \rangle$ for $i \in [0,n]$.

$$\mathrm{SD}(\overline{s}^*, p) \triangleq \begin{cases} \mathrm{SD}(\overline{s'}^*, p) \cdot pa_n & \text{if } M(p, g_n) \wedge n > 0 \\ \mathrm{SD}(\overline{s'}^*, p) & \text{if } \neg M(p, g_n) \wedge n > 0 \\ \varepsilon & \text{otherwise} \end{cases}$$

where $pa_n = \left\langle D[a_n^*, K_{g_n}], p_n, g_n, h_n \right\rangle$ and ε is the empty sequence.

In the case where the peer belongs to the group advertised ($M(p, g_n)$), the last action is decrypted using the group key ($D[a_n^*, K_{g_n}]$) and the resulting tuple is included in the decrypted sequence. Otherwise, when the peer does not belong to the group ($\neg M(p, g_n)$), the entire tuple is discarded from the sequence.

F. Checking the lifecycle

A peer p verifies the lifecycle of the document based on his groups. To do so, the peer first computes $\overline{s}_p = \mathrm{SD}(\overline{s}^*, p)$, then ensures that $\delta_p(\overline{s}_p) = \top$. $\overline{s}_p$ is the sequence that p can decrypt based on his groups, while δ_p is the lifecycle p can verify (defined in Section IV-A).

G. Checking the document

The purpose of the digest is to provide the receiver of a document a guarantee about the authenticity of the peer-action sequence that it contains. This sequence, in turn, can be used to check that the document being passed is genuine and has not been manipulated.

Given the decrypted peer-action section for p, $\overline{s}_p = \mathrm{SD}(\overline{s}^*, p) = (\langle a_0, p_0, g_0, h_0 \rangle, \ldots, \langle a_k, p_k, g_k, h_k \rangle)$. Since the peer-action sequence can omit some encrypted parts, we have $|\overline{s}_p| \leq |\overline{s}^*|$. Starting from the base document $d_\emptyset$ it is possible to

Group	e_1	e_2	e_3	e_4	e_5
t	X	X	-	-	X
t'	-	-	X	X	X
m	-	-	-	-	X

Table I: Groups and Membership

Peer	$\bar{s}_p$	δ_p
e_1, e_2	$(y_1, n_2, send)$	δ_t
e_3, e_4	$(forward, y_3, y_4)$	$\delta_{t'}$
e_5	$(y_1, n_2, send, com, forward, y_3, y_4)$	$\delta_t \wedge \delta_{t'} \wedge \delta_m$

Table II: Decryption and Checking

compute the new document $d = f_{a_k}(f_{a_{k-1}} \cdots (f_{a_1}(d_\emptyset)) \cdots)$, and compare it with the document being passed. In other words, it is possible for a peer to "replay" the complete sequence of actions, starting from the empty document, and to compare the result of this sequence to the actual document. Since some actions are hidden from the peer, it is not possible to reconstruct the entire document unless p is in all groups (in which case $\bar{s}_p = \bar{s}$). However it is possible to verify a part of the document if we consider the following assumptions on the specification:

1) The data in the document is partitioned into pair-wise disjoint sets $D = \bigcup_{g \in G}(D_g)$.
2) For each action a appearing in a lifecycle δ_g, f_a either modifies data in D_g or no data at all (does not modify the document).

With these assumptions, people in the group can "replay" only the data relevant to the group, since group actions do not interfere with it. Actions associated with functions that do not modify the document could be used to synchronize the various groups. One could consider that each set of fields is encrypted with the group key, so as to not be visible for other groups.

Note that in some cases, knowledge of the peer-action sequence and of the empty document is sufficient to reconstruct the complete document without the need to pass it along. In such cases, only exchanging the sequence and the digest is necessary. However, there exist situations where this does not apply —for example, when the document is a physical object that has to be passed from one peer to the next (as in the case of a metro card), or when the data subject to modification is a subset of all data carried in the document.

H. Full Example

We consider a company consisting of two teams t and t' with two employees in each, respectively e_1, e_2 and e_3, e_4. Additionally we consider the manager e_5 belonging to the group m. Our document is a form which requires voting on a project, the project requires at least one person per team and both teams to vote `yes` to pass. The groups are the summarized in Table I.

We partition the document into three sets $S = S_t \cup S_{t'} \cup S_m$ with $S_t = \{f_1, f_2\}$, $S_{t'} = \{f_3, f_4\}$ and $S_m = \{f_5\}$. And define the following actions: $A = \{y_i, n_i \mid i \in [1,4]\} \cup \{com, send, forward\}$ Such that y_i (n_i) is associated with the function that writes `yes` (resp. `no`) into field f_i for an employee e_i. *send* specifies an action to send the result to the manager. The manager then processes it by adding comments in f_5 using the action *com*, and forwards it to the other group with the action $forward$.

We begin by defining the specification. We use the predicate vote$(\bar{s}, y, y')$ to abstract the voting procedure of two members. The vote succeeds if either y or y' appear only once in the sequence prior to *send*. Given a sequence $\bar{s} = (\langle a_0, p_0, g_0, h_0 \rangle, \ldots, \langle a_n, p_n, g_n . h_n \rangle)$, we define:

$$
\begin{aligned}
\text{after}(\bar{s}, y, x) &\equiv \forall i \in [0,n] : (a_i = y) \rightarrow \exists j < i : a_j = x \\
\text{vote}(\bar{s}, y, y') &\equiv \forall i \in [0,n] : \\
&\quad (a_i = send) \rightarrow (\exists j < i : a_j \in \{y, y'\}) \\
&\quad \wedge (a_i = y) \rightarrow (\nexists j \neq i : a_j = y) \\
&\quad \wedge (a_i = y') \rightarrow (\nexists k \neq i : a_k = y')
\end{aligned}
$$

$$
\delta_t(\bar{s}) \equiv \begin{cases} \top & \text{if } vote(\bar{s}, y_1, y_2) \\ & \quad \wedge (\text{after}(\bar{s}, send, y_2) \vee \text{after}(\bar{s}, send, y_1)) \\ \bot & \text{otherwise} \end{cases}
$$

$$
\delta_{t'}(\bar{s}) \equiv \begin{cases} \top & \text{if } vote(\bar{s}, y_3, y_4) \wedge \text{after}(\bar{s}, y_3, forward) \\ & \quad \wedge \text{after}(\bar{s}, y_4, forward) \\ \bot & \text{otherwise} \end{cases}
$$

$$
\delta_m(\bar{s}) \equiv \begin{cases} \top & \text{if after}(\bar{s}, com, send) \wedge \text{after}(\bar{s}, forward, com) \\ \bot & \text{otherwise} \end{cases}
$$

In this example scenario, employee e_1 votes `yes` wile employee e_2 votes `no` the sequence is now: $\bar{s}^* = (\langle E[y_1, S_t], e_1, t, h_1 \rangle, \langle E[n_2, S_t], e_2, t, h_2 \rangle)$. Employee e_1 then sends the form to the manager s adding: $\langle E[send, S_t], e_1, t, h_3 \rangle$ The manager comments, then signs it but on behalf of the manager group: $\langle E[com, S_m], e_5, m, h_4 \rangle$ The manager forwards it to the second team on behalf of the second team: $\langle E[forward, S_{t'}], e_5, t', h_5 \rangle$. The second team then votes with both accepting the sequence is then appended: $(\langle E[y_3, S_{t'}], e_3, t', h_6 \rangle, \langle E[y_4, S_{t'}], e_4, t', h_7 \rangle)$. We show the decrypted peer-action sequence for each peer in Table II (for brevity we display only the actions).

In this case, the fields for each group are separate, each group can replay their own event and check if the document matches them. The only person in this case capable of verifying the entire document is e_5 as they are in all groups. The groups serve to hide the actions of specific individuals. In this case the teams cannot know which individual voted a "`no`" from the other team, even-though they know that they voted. Voting in this case is unanimous. It is also possible to encrypt the specification itself, that is each lifecycle δ_g can be encrypted with the key of the group S_g. Thus it is possible to hide specification from the non-group members.

V. Implementation and Discussion

To illustrate the concepts introduced in this paper, we applied them to the specific scenario where an artifact is a dynamic, fillable PDF form. In this context, the various fields of the form constitute the document's data, which can be filled and modified by various peers. A special, hidden form field is included to the document, which is intended to contain the

peer-action sequence reflecting the document's modification history.[2]

We implemented *Artichoke*, a command-line front-end to inject and manage peer-action sequences into these forms. Artichoke uses LaTeX to generate forms with various input and an empty peer-action sequence. It also uses pdftk[3] to extract and manipulate form data in the background. Although Artichoke is intended as a proof-of-concept implementation with minimal user-friendliness, it is fully functional and its source code is public available under the GNU GPL.[4] The current implementation supports a slightly simplified version of peer-action sequences, where a single group exists, but peers in the group each have their own public/private key pair to stamp their actions.

A. Usage

Currently, Artichoke supports the three main operations on a document, namely filling, examining and checking the peer-action sequence of a form.

A first operation is to fill a form, which consists in writing (or overwriting) one or more form fields with specified values. In our context, filling a form also involves updating the peer-action sequence contained in that form to include the modification action and peer information related to that action.

A second operation is to examine the contents of a form; This will print the current value of all the form's fields, and display a summary of the peer-action sequence contained in the document, which will look like this:

```
Form fields
-----------
F1:     baz
F2:     bar

Peer-action sequence
--------------------
Alice   W|F1|foo       Rm/MRSzK...oYpROg0=
Bob     W|F2|bar       kEvrkC+e...bX4NO1w=
Carl    W|F1|baz       F3UYg+n1.../YPs3/k=
```

The peer-action sequence shows that Alice first wrote "foo" to field F1, then Bob wrote "bar" to field F2, then Carl overwrote F1 with "baz". The rightmost column is a shortened version of the digest string for each event.

The last operation that can be done with Artichoke is to validate the contents and history of a form. The policy is currently specified through user-defined PHP code, by implementing a special function called `check_policy` that receives as its input the peer-action sequence of the current document. Hence the enforcement of a policy is not tied to any particular specification language, provided it can be expressed in terms of the contents of peer-action sequence only.

[2]Note that this field is only made invisible for the sake of readability; its hidden nature has nothing to do with protecting it from tampering.

[3]http://pdftk.org

[4]https://github.com/liflab/artichoke

B. Resource Consumption

Equipped with this implementation, we proceeded to perform tests intended to measure the computational resources required in a typical use-case scenario. In particular, we want to determine whether the repeated application of encryption and hashing induces a reasonable cost, in terms of both time and space, as the history of a document lengthens over time.

The tests were implemented using the ParkBench testing framework.[5] We first generated an empty PDF form with a single input field F. Using the commands described above, the test script was then instructed to repeatedly overwrite F with a dummy value on behalf of some peer. This had for effect of creating a set of PDF files containing a peer-action sequence of increasing length.

The first factor we measured is the running time for appending a new action to an existing document. This is shown in Figure 4a. One can see that the running time increases linearly with the number of write operations. We can deduce from this graph that it takes approximately one second to perform a single write operation. Note that in this proof-of-concept implementation, this is done through a chain of calls to command-line software; in particular, each write operation requires calling pdftk a first time to dump the file's data fields, processing these fields, and calling pdftk a second time to create a new PDF file with updated data fields. Better performance could be easily achieved by directly implementing all these operations in a single program, rather than relying on costly external calls.

The second factor we measured is the time required to simply *check* an existing document without modifying it; this is shown in Figure 4b. It takes between 200 and 700 ms to check an entire peer-action sequence; repeated experiments could not highlight any trend in the running time as the sequence lengthens. Nevertheless, this seems to indicate that read/decrypt operations are much quicker to perform than write/encrypt ones.

The final factor we measured is the size of the document for increasing lengths of a peer-action sequence; this is shown in Figure 4c. As expected, the size of the sequence grows linearly with the number of operations applied to the document, indicating that each element of the sequence requires constant space. In the current implementation of Artichoke, this space amounts to roughly 40 bytes.

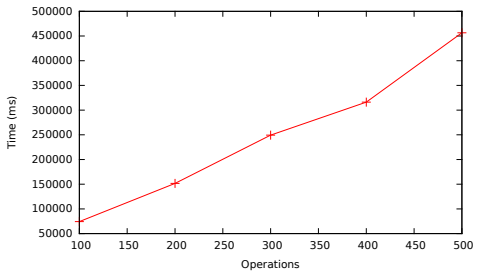
C. Discussion

Overall, the positive results obtained with the current implementation of Artichoke illustrate the potential of peer-action sequences to effectively encode a document's history so that lifecycle constraints can be verified on it at any moment. We mention in the following a few discussion points regarding the current system.

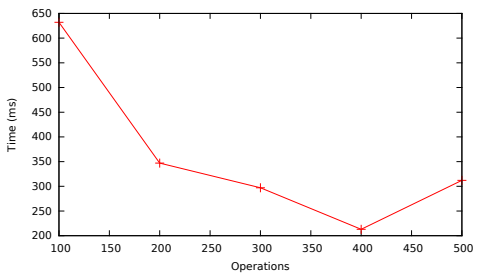
1) Stateful vs. stateless peers: As such, the peers in the exchange can be completely *stateless*: they are not required to persist any information between accesses to a document, apart from their public/private key pair.[6] All the history and the

[5]https://sylvainhalle.github.io/ParkBench

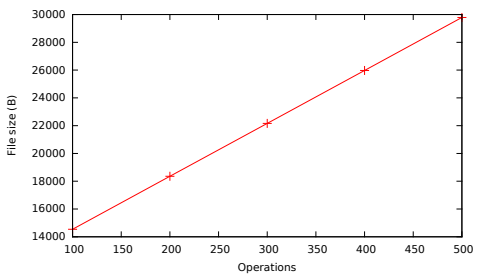
[6]They must also remember the lifecycle function being enforced; however this could even be saved within the document and encrypted with their private key. In any case the function is likely to be the same for all documents, and could in some cases be hard-coded into the peers' read-only memory.



(a) Writing to the document



(b) Checking the document



(c) Digest size

Figure 4: Running time of Artichoke on PDF documents with a peer-action sequence of increasing length: (a) to write the to document; (b) to check a document; (c) File size of the PDF document with a peer-action sequence of increasing length.

verification of the lifecycle can be reconstructed from the empty document at any time. However, a stateful peer can save on processing time: for each document, such a peer can save the digest and state of the document each time it receives it. Upon receiving it another time, it only requires to invert the digest and check the document's contents up to its last locally-saved state. (This is possible, since the probability for a tampered document to yield the same n-th digest as the original is very small.) This way, each element of the peer-action sequence requires processing only once.

2) Space requirements: In addition to the document's contents, storage space is required to hold the peer-action sequence, whose size is proportional to the length of the history. Note that in the general setting, this sequence cannot simply be trimmed of its first events after "long enough", as a peer could use this facility to cover up a fraudulent manipulation of the document. The question remains open whether peers can be given any freedom in erasing prefixes of the history without the possibility of misuse.

3) Enforcement: In the proposed system, the enforcement of lifecycle constraints is indirect. Any peer can tamper with the contents of the document, with its history, or perform modifications that violate the lifecycle requirements. Likewise, any peer can choose to accept such a tampered document, modify it and pass it on to other peers. However, our approach makes sure that anyone with knowledge of the peers' public keys (including peers external to the exchange) can check at any time whether such misuses occurred, as well as pinpoint what peers have been faulty or complacent.

4) Duplication: In some situations, the document can be duplicated. Therefore, a peer can receive a document, modify it in two different ways, and pass it on to two different peers. Our proposed approach will still ensure that each copy will follow a compliant lifecycle, but the uniqueness of each document cannot be ensured. However, since our approach allows the specification of a lifecycle for a document, conditions can be added to this lifecycle so that uniqueness is guaranteed. One simple (and relatively restrictive) condition could be that at any point, the possible sender for the next action is always unique (and would henceforth detect if the same document is sent twice). Determining conditions for uniqueness is outside the scope of this work.

VI. Conclusion and Future Work

In this paper, we have shown how the lifecycle of an artifact can be effectively stored within the document itself, using the concept of peer-action sequences. Moreover, this sequence can be protected from tampering through an appropriate use of public-key encryption and hashing. This provides at the same time a mechanism for enforcing different read-write access permissions to various parts of the document, depending on the *group* a peer belongs to. Experiments have shown that manipulating these sequences does not impose an undue burden in terms of computing resources, and that the space required to store a sequence within a document increases linearly with the number of modifications made to it.

The main advantage of peer-action sequences, over existing lifecycle compliance approaches, is the fact that compliance can be checked on-the-fly and at any moment on a document that can be freely exchanged between peers. Peers do not need to be statically verified prior to any interaction, and the document is not required to be accessed from a single point in order to enforce compliance. This presents the potential of greatly simplifying the implementation of artifact-centric workflows, by dropping many assumptions that must be fulfilled by current systems. Taken to its extreme, lifecycle policies can even be verified without resorting to any workflow management system at all: as long as documents are properly stamped by everybody, the precise way they are exchanged (e-mail, file copying, etc.) is irrelevant.

Technically speaking, the next step of this work will be to port Artichoke so that it stores its peer-action sequences into a data field compliant with the Extensible metadata platform (XMP) standard [1]. This will allow peer-action sequences to be stored not only in PDF documents, but also in any media type that supports XMP: JPEG images, MP3 files, HTML documents, etc. One could hence imagine lifecycle policies for types of documents not traditionally considered by the

business process community —such as restrictions on the way image files can be manipulated. In the case of forms, the filling, stamping and compliance checking of PDF files with respect to a peer-action sequence could be implemented directly into the graphical user interface of a PDF reader, and become a seamless process that could be executed by a user in a single button click.

On the formal side, a number of possible extensions and open questions also arise. For example, can we replace the current history by some token whose size over time remains bounded by a constant? Similarly, could we enforce proper usage by rendering the document unreadable if improperly modified? This way a peer would not even need to replay the history: simply trying to read the document would reveal a problem. The enforcement of constraints across multiple documents in the same lifecycle is also an open issue; the use of synchronization signals between peers, borrowed from decentralized runtime monitoring, could prove a promising solution. Finally, the question of uniqueness of documents also needs to be studied. In its current incarnation, the proposed system allow artifacts to be duplicated, yet enforces that all copies must follow a valid lifecycle.

REFERENCES

[1] Graphic technology – extensible metadata platform (XMP) specification – part 1: Data model, serialization and core properties. Technical Report ISO Standard 16684-1, 2012.

[2] A. A. Ataullah and F. W. Tompa. Business policy modeling and enforcement in databases. *PVLDB*, 4(11):921–931, 2011.

[3] A. K. Bauer and Y. Falcone. Decentralised LTL monitoring. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *Lecture Notes in Computer Science*, pages 85–100. Springer, 2012.

[4] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In G. Alonso, P. Dadam, and M. Rosemann, editors, *BPM*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–304. Springer, 2007.

[5] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corporation, 1977.

[6] N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems 2011*, volume 6542, pages 73–86. Springer, 2011.

[7] N. Bielova, F. Massacci, and A. Micheletti. Towards practical enforcement theories. In *Proceedings of The 14th Nordic Conference on Secure IT Systems*, volume 5838 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag Heidelberg, 2009.

[8] W. Boebert and R. Kain. A practical alternative to hierarchical integrity policies. In *S&P*, 1985.

[9] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *S&P*, pages 206–214. IEEE Computer Society, 1989.

[10] D. Calvanese, G. De Giacomo, R. Hull, and J. Su. Artifact-centric workflow dominance. In L. Baresi, C. Chi, and J. Suzuki, editors, *ICSOC-ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 130–143, 2009.

[11] D. Calvanese, M. Montali, M. Estañol, and E. Teniente. Verifiable UML artifact-centric business process models. In J. Li, X. S. Wang, M. N. Garofalakis, I. Soboroff, T. Suel, and M. Wang, editors, *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, CIKM 2014, Shanghai, China, November 3-7, 2014*, pages 1289–1298. ACM, 2014.

[12] C. Colombo and Y. Falcone. Organising LTL monitors over distributed systems with a global clock. In B. Bonakdarpour and S. A. Smolka, editors, *RV*, volume 8734 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2014.

[13] Y. Falcone, T. Cornebize, and J. Fernandez. Efficient and generalized decentralized monitoring of regular languages. In E. Ábrahám and C. Palamidessi, editors, *FORTE*, volume 8461 of *Lecture Notes in Computer Science*, pages 66–83. Springer, 2014.

[14] D. Ferraiolo and D. Kuhn. Role-based access control. In *S&P*, page 554–563, October 1992.

[15] P. W. L. Fong. Access control by tracking shallow execution history. In *S&P*, pages 43–55, 2004.

[16] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In B. J. Krämer, K. Lin, and P. Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2007.

[17] P. Gonzalez, A. Griesmayer, and A. Lomuscio. Verifying gsm-based business artifacts. In C. A. Goble, P. P. Chen, and J. Zhang, editors, *2012 IEEE 19th International Conference on Web Services, Honolulu, HI, USA, June 24-29, 2012*, pages 25–32. IEEE Computer Society, 2012.

[18] S. Hallé. Cooperative runtime monitoring. *Enterprise IS*, 7(4):395–423, 2013.

[19] S. Hallé and R. Villemaire. Runtime enforcement of web service message contracts with data. *IEEE Trans. Services Computing*, 5(2):192–206, 2012.

[20] S. Hallé, R. Villemaire, and O. Cherkaoui. Specifying and validating data-aware temporal web service properties. *IEEE Trans. Software Eng.*, 35(5):669–683, 2009.

[21] B. B. Hariri, D. Calvanese, G. De Giacomo, R. De Masellis, and P. Felli. Foundations of relational artifacts verification. In S. Rinderle-Ma, F. Toumani, and K. Wolf, editors, *BPM*, volume 6896 of *Lecture Notes in Computer Science*, pages 379–395. Springer, 2011.

[22] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F. T. Heath, S. Hobson, M. H. Linehan, S. Maradugu, A. Nigam, P. N. Sukaviriya, and R. Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In D. M. Eyers, O. Etzion, A. Gal, S. B. Zdonik, and P. Vincent, editors, *DEBS*, pages 51–62. ACM, 2011.

[23] S. Kumaran, R. Liu, and F. Y. Wu. On the duality of information-centric and activity-centric models of business processes. In Z. Bellahsene and M. Léonard, editors, *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 32–47. Springer, 2008.

[24] V. Künzle and M. Reichert. Philharmonicflows: towards a framework for object-aware process management. *J. of Software Maintenance*, 23(4):205–244, 2011.

[25] A. Meyer, L. Pufahl, D. Fahland, and M. Weske. Modeling and enacting complex data dependencies in business processes. In F. Daniel, J. Wang, and B. Weber, editors, *BPM*, volume 8094 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2013.

[26] R. R. Mukkamala, T. T. Hildebrandt, and J. B. Tøth. The resultmaker online consultant: From declarative workflow management in practice to LTL. In M. van Sinderen, J. P. A. Almeida, L. F. Pires, and M. Steen, editors, *EDOCW*, pages 135–142. IEEE Computer Society, 2008.

[27] P. Nandi, D. Koenig, S. Moser, R. Hull, V. Klicnik, S. Claussen, M. Kloppmann, and J. Vergo. Data4BPM, part 1: Introducing business entities and the business entity definition language (BEDL), April 2010.

[28] A. Nigam and N. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, 2003.

[29] K. Ouafi and S. Vaudenay. Pathchecker: an RFID application for tracing products in supply-chains. In *RFIDSec*, pages 1–14, 2009.

[30] A. E. K. Sobel and J. Alves-Foss. A trace-based model of the Chinese wall security policy. In *Proc. of the 22nd National Information Systems Security Conference*, 1999.

[31] R. Vaculín, R. Hull, T. Heath, C. Cochran, A. Nigam, and P. Sukaviriya. Declarative business artifact centric modeling of decision and knowledge intensive business processes. In *EDOC*, pages 151–160. IEEE Computer Society, 2011.

[32] X. Zhao, J. Su, H. Yang, and Z. Qiu. Enforcing constraints on life cycles of business artifacts. In W. Chin and S. Qin, editors, *TASE*, pages 111–118. IEEE Computer Society, 2009.