On the Satisfiability of Workflows with Release Points

Jason Crampton
Royal Holloway, University of London
Egham, United Kingdom
jason.crampton@rhul.ac.uk

Gregory Gutin
Royal Holloway, University of London
Egham, United Kingdom
g.gutin@rhul.ac.uk

Rémi Watrigant
Inria Sophia Antipolis
Sophia Antipolis, France
remi.watrigant@inria.fr

ABSTRACT

There has been a considerable amount of interest in recent years in the problem of workflow satisfiability, which asks whether the existence of constraints in a workflow specification means that it is impossible to allocate authorized users to each step in the workflow. Recent developments have seen the workflow satisfiability problem (WSP) studied in the context of workflow specifications in which the set of steps may vary from one instance of the workflow to another. This, in turn, means that some constraints may only apply to certain workflow instances. Inevitably, WSP becomes more complex for such workflow specifications. In this paper, we present the first fixed parameter algorithms to solve WSP for workflow specifications of this type. Moreover, we significantly extend the range of constraints that can be used in workflow specifications of this type.

KEYWORDS

workflow satisfiability, release points, workflow composition, xor-branching

ACM Reference format:
Jason Crampton, Gregory Gutin, and Rémi Watrigant. 2017. On the Satisfiability of Workflows with Release Points. In *Proceedings of SACMAT'17, Indianapolis, IN, USA, June 21-23, 2017,* 11 pages.
DOI: http://dx.doi.org/10.1145/3078861.3078866

1 INTRODUCTION

Many businesses use computerized systems to manage their business processes. A common example of such a system is a workflow management system which is responsible for the co-ordination and execution of steps in a business process. A business process may be executed many times and by different users. However, the structure of the process is fixed and may be defined by a set of steps that must be performed in a particular sequence. In addition, we may wish to impose some form of access control on the execution of a business process, limiting which users may perform which steps. This control may take the form of an authorization policy, which defines which users are authorized to perform which steps, and authorization constraints, which limit the combinations of users that may perform certain sets of steps in the business process. A simple form of constraint could prohibit the same user from performing two (or more) particular security-sensitive steps.

SACMAT'17, Indianapolis, IN, USA

DOI: http://dx.doi.org/10.1145/3078861.3078866

The structure of a business process or workflow need not be linear. There may be subprocesses that can be performed in parallel, or there may be subprocesses that are mutually exclusive (and only one of the subprocesses is executed in a particular instance of the workflow). Thus, the steps executed in a workflow may vary from one instance to another. Moreover, there may be constraints that only apply when certain subprocesses are executed. Basin, Burri and Karjoth introduced a mechanism for modeling such constraints using *release points* [3]. Informally, release points allow a constraint to be "switched off" when some given points of an instance workflow are reached. In particular, when different release points are located in different mutually exclusive subprocesses, it is possible to encode conditional constraints.

Determining whether a workflow specification is satisfiable – in the sense that there exists an allocation of authorized users to steps such that all constraints are satisfied – is an important question. An algorithm for deciding the so-called *workflow satisfiability question* (WSP) is important from the point of view of static analysis of workflow specifications and for workflow management systems in which users select which steps to execute [10, Section 2.2]. However, most work on WSP has assumed that the set of steps is the same for all workflow instances. The exception is the work of Crampton and Gutin [10], who introduced a simple language for workflow composition, to model workflows with parallel and exclusive-or branching. However, their work does not consider the effect of release points on satisfiability. Conversely, the work of Basin *et al.* does not provide an exact algorithm for solving the enforcement process existence (EPE) problem, a problem essentially equivalent to WSP in the presence of release points. The heuristic algorithm developed by Basin *et al.* to solve the EPE problem produces good results "when the set of users is large and the static authorizations are equally distributed among them". It is unclear whether the requirement that static authorizations be equally distributed is likely to hold in practical settings.

In this paper, we extend the work of Crampton and Gutin [10], who introduced a simple language for composing workflows and solving WSP for workflows specified using this language, to incorporate release points. We then extend the definition of constraint satisfaction, relative to a particular execution of the steps in such a workflow, in the presence of release points. Finally, we develop fixed-parameter algorithms that solve WSP for workflows incorporating release points, thereby providing the first results in this area. Moreover, our notion of constraints with release points is a significant generalization of that used by Basin *et al.*

In the remainder of this section we introduce relevant notation, terminology and background material. In Section 2, we define the notion of a compositional workflow with release points, extending the notion of constraint satisfaction accordingly. In Section 3, we describe our method for solving WSP and provide an analysis of its

complexity. We briefly discuss related work in Section 4. The paper concludes with a summary of our contributions and our ideas for future research in this area.

1.1 Notation and Terminology

A directed graph (*digraph* for short) is a pair $G = (V, E)$, where V is the set of vertices, and $E \subseteq V \times V$ is the set of edges. A directed acyclic graph (*DAG* for short) is a digraph which does not contain any directed cycle, *i.e.* no sequence $(u_0, u_1 \ldots, u_{k-1}, u_0)$ such that each pair of consecutive vertices belongs to E. For $u \in V$, we define the *in-neighborhood* of u to be the set $N^-(u) = \{t \in V | (t, u) \in E\}$; the *in-degree* of u is the size of its in-neighborhood. Similarly, the *out-neighborhood* of u is the set $N^+(u) = \{w \in V | (u, w) \in E\}$ and the *out-degree* of u is the size of its out-neighborhood. A vertex of in-degree 0 is called a *source*, while a vertex of out-degree 0 is called a *sink*. For $S \subseteq V$, we denote by $G[S]$ the *induced subgraph* $(S, E \cap (S \times S))$. By abuse of notation, we will sometimes write $G \setminus S$ as a shortcut for $G[V \setminus S]$. For more information about graphs and DAGs, we refer the reader to [2, 13].

Sometimes, it is convenient to represent a DAG with a partial order on its vertices. Indeed, we may write $u \leq v$ for $u, v \in V$ whenever $u = v$ or there exists a directed path from u to v. By extension, we may write $u < v$ if $u \leq v$ and $u \neq v$.

For any positive integer n, let $[n] = \{1, \ldots, n\}$. An ordered sequence $\sigma = (v_1, \ldots, v_q)$ of distinct vertices of V is called a *linear subextension* of G iff for every $i, j \in [q]$, $v_i \leq v_j$ implies $i \leq j$. If σ contains all vertices of V, then we say that σ is a *linear extension* of G.

Many decision problems take several parameters as input. It can be instructive to consider how the complexity of the problem may change if we assume one or more of those parameters is small relative to the others. The purpose of multivariate analysis of the complexity of a problem is to obtain efficient algorithms when the chosen parameters take small values in practice. We say that a decision problem is *fixed-parameter tractable* (FPT) if there exists an algorithm that decides if an instance is positive in $O(f(k)p(n))$ time for some computable function f and some polynomial p, where n denotes the size of an instance, and k is a parameter of the instance. Accordingly, we will call such an algorithm an *FPT algorithm*. For more details about parameterized complexity, we refer the reader to the monographs of Downey and Fellows [14] and Cygan *et al.* [12].

1.2 The workflow satisfiability problem

A *workflow specification* is defined by a directed acyclic graph $G = (S, E)$, where S is the set of steps to be executed, and $E \subseteq S \times S$ defines a partial ordering on the set of steps in the workflow, in the sense that $(s_1, s_2) \in E$ means that step s_1 must be executed before s_2 in every instance of the workflow. Note that the order is not required to be total, so the exact sequence of steps may vary from instance to instance. In addition, we are also given a set of users U and an *authorization policy* $A \subseteq S \times U$, where $(s, u) \in A$ means that user u is authorized to execute step s. A workflow specification $G = (S, E)$ together with an authorization policy is called a *workflow schema*. Throughout the paper, we will assume that for every step $s \in S$, there exists some user $u \in U$ such that $(s, u) \in A$.

A workflow *constraint* (T, Θ) limits the users that are allowed to perform a set of steps T in any execution of the workflow. In particular, Θ identifies authorized (partial) assignments of users to steps in T, i.e. Θ is a set of functions from T to U. A (partial) plan is a function $\pi : S' \to U$, where $S' \subseteq S$. A plan $\pi : S \to U$ represents an allocation of steps to users. The workflow satisfiability problem (WSP) is concerned with the existence or otherwise of a plan that is authorized and satisfies all constraints.

More formally, let $\pi : S' \to U$, where $S' \subseteq S$, be a plan. Given $T \subseteq S'$, we write $\pi|_T$ to denote the function π restricted to domain T; that is $\pi|_T : T \to U$ is defined by $\pi|_T(s) = \pi(s)$ for all $s \in T$. Then we say $\pi : S' \to U$ *satisfies* a workflow constraint (T, Θ) if $T \not\subseteq S'$ or $\pi|_T \in \Theta$.

In practice, we do not define a constraint by giving the family of functions Θ extensionally, as the size of such set might be exponential in the number of users and steps. Instead, we will assume that constraints have "compact" descriptions, in the sense that it takes polynomial time to test whether a given plan satisfies a constraint. This is a reasonable assumption, as all constraints of relevance in practice satisfy such a property. For instance, the two most well-known constraints are perhaps *binding-of-duty (BoD)* and *separation-of-duty (SoD)*. The *scope* of these constraints is binary: a plan π satisfies a BoD constraint $(\{s_1, s_2\}, =)$ iff $\pi(s_1) = \pi(s_2)$; and π satisfies an SoD constraint $(\{s_1, s_2\}, \neq)$ iff $\pi(s_1) \neq \pi(s_2)$. A natural generalization of these constraints are atmost and atleast constraints, in which the scope may be of arbitrary size, and the definition of such constraints includes an additional integer k. Given $T \subseteq S$, a plan satisfies atmost(T, k) (resp. atleast(T, k)) iff $|\pi(T)| \leq k$ (resp. $|\pi(T)| \geq k$).

User-independent constraints generalize all these forms of constraints [6]. Informally, such a constraint limits the execution of steps in a workflow, but is indifferent to the particular users that execute the steps. More formally, a constraint (T, Θ) is user-independent if whenever $\theta \in \Theta$ and $\psi : U \to U$ is a permutation then $\psi \circ \theta \in \Theta$ (where $\circ$ denotes function composition). A separation of duty constraint, on two steps for example, simply requires that two *different* users execute the steps, not that, say, Alice and Bob (in particular) must execute them. Similarly, a binding of duty constraint on two steps only requires that the *same* user executes the steps. More generally, atleast and atmost constraints are user-independent. It appears most constraints that are useful in practice are user-independent: all constraints defined in the ANSI-RBAC standard [1], for example, are user-independent.

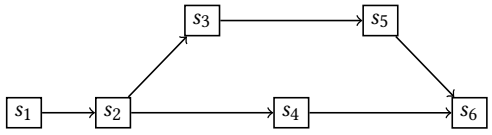
A *constrained workflow authorization schema* is a tuple $(G = (S, E), U, A, C)$, where (G, U, A) is a workflow schema, and C is a set of constraints. We say that a plan $\pi : S \to U$ is *authorized* if $(s, \pi(s)) \in A$ for every $s \in S$, and we say that π is *valid* if it is authorized and if it satisfies all $c \in C$. We are now ready to introduce the Workflow Satisfiability Problem, as defined by Wang and Li [25]:

> Workflow Satisfiability Problem (WSP)
> *Input:* A constrained workflow authorization schema $W = (G = (S, E), U, A, C)$
> *Question:* Is there a valid plan $\pi : S \to U$?

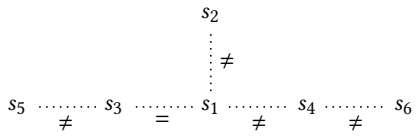
We present as a running example a simple purchase-order workflow [8] in Figure 1. We will extend this example in subsequent

sections in order to illustrate the concepts introduced in this paper. In the first step of this workflow, the purchase order is created and approved (and then dispatched to the supplier). The supplier will submit an invoice for the goods ordered, which is processed by the create payment step. When the supplier delivers the goods, a goods received note (GRN) must be signed and countersigned. Only then may the payment be approved and sent to the supplier. Observe that this workflow specification contains parallel branches, in the sense that the processing of both s_3 and s_4 must occur before s_6, but the relative ordering of s_3 and s_4 is of no importance. We will extend this example to include mutually exclusive branches.

The workflow specification also includes constraints (each having binary scope), mainly in order to reduce the possibility of fraud. Such constraints may be depicted as an undirected, labeled graph, in which the vertices represent steps and edges denote constraints, as illustrated in Figure 1(b). One requirement, for example, is that the steps to create and approve a purchase order are executed by different users. We will extend the example to include constraints having release points.



(a) Ordering on steps



(b) Constraints

s_1	create purchase order
s_2	approve purchase order
s_3	sign GRN
s_4	create payment
s_5	countersign GRN
s_6	approve payment
≠	different users must perform steps
=	same user must perform steps

(c) Legend

Figure 1: A simple constrained workflow for purchase order processing

2 COMPOSITIONAL WORKFLOWS AND RELEASE POINTS

We now extend the definitions of workflow specification and workflow schema to a compositional variant. We also extend the constraints model to introduce release points.

2.1 Workflow composition

We now introduce a convenient way to represent situations where, at some points of a workflow execution, one would like to branch into several subworkflows independently, a notion also known as *OR-forks* [23] or *exclusive gateways* [26]. To that end, we use the model defined by Crampton and Gutin [10] called *Workflow Composition*.

A *compositional workflow specification* is defined recursively using three operations: serial composition, parallel branching and xor branching. Like a "classical" workflow specification, it can be represented as a DAG $G = (V, E)$. However, in the case of a compositional workflow, not all vertices represent steps. In addition to the set of (classical) steps, V also contains R, the set of *release points*, and O, the set of *orchestration points*. Orchestration points will be introduced shortly. Release points limit actions of constraints by restricting their scopes and will be introduced in Section 2.3. We will sometimes directly define a compositional workflow specification as $G = (S \cup R \cup O, E)$.

The DAG of a compositional workflow always contains two special orchestration points: a source vertex α, called *input* and a sink vertex ω, called *output*. Moreover, an atomic compositional workflow (*i.e.* the base case for constructing such a workflow) is composed of a single step or release point v, and can be represented by the DAG $G = (\{\alpha, v, \omega\}, \{(\alpha, v), (v, \omega)\})$. Given two compositional workflows $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ with respective input and output vertices α_1, ω_1 and α_2, ω_2, respectively, we may construct new compositional workflows using serial composition, and parallel and xor branchings, denoted by $G_1; G_2$, $G_1 \parallel G_2$ and $G_1 \otimes G_2$, respectively. We assume that $V_1 \cap V_2 = \emptyset$.

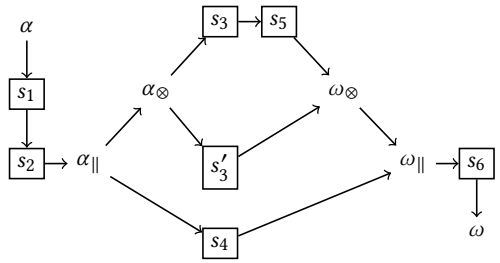
For *serial composition*, all the steps in G_1 must be completed before the steps in G_2. Hence, the DAG of $G_1; G_2$ is formed by taking the union of V_1 and V_2, the union of E_1 and E_2, and the addition of a single edge from ω_1 to α_2. Thus, α_1 (resp. ω_2) is the input (resp. output) vertex of $G_1; G_2$.

For *parallel composition*, the execution of the steps in G_1 and G_2 may be interleaved. Hence, the DAG of $G_1 \parallel G_2$ is formed by taking the union of V_1 and V_2, the union of E_1 and E_2, the addition of new input and output vertices $\alpha_\parallel$ and $\omega_\parallel$, and the addition of edges $(\alpha_\parallel, \alpha_1), (\alpha_\parallel, \alpha_2), (\omega_1, \omega_\parallel)$ and $(\omega_2, \omega_\parallel)$. This form of composition is sometimes known as an *AND-fork* [23] or a *parallel gateway* [26].

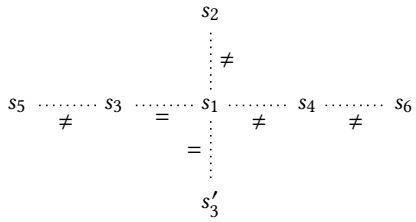
In both serial and parallel compositions, all steps in G_1 and G_2 are executed. In *xor composition*, either the steps in G_1 are executed or the steps in G_2, but not both. In other words, xor composition represents non-deterministic choice in a workflow specification. The DAG $G_1 \otimes G_2$ is formed by taking the union of V_1 and V_2, the union of E_1 and E_2, the addition of new input and output vertices $\alpha_\otimes$ and $\omega_\otimes$, and the addition of edges $(\alpha_\otimes, \alpha_1), (\alpha_\otimes, \alpha_2), (\omega_1, \omega_\otimes)$ and $(\omega_2, \omega_\otimes)$. Given $G_1 \otimes G_2$, we will say that every pair of vertices $(v, v') \in V_1 \times V_2$ are *exclusive*. We say that a compositional workflow is *xor-free* if it can be constructed with only serial and parallel operations.

For the sake of readability, we will sometimes simplify the representation of a compositional workflow by replacing an orchestration point having a single in-neighbor u and a single out-neighbor v by the edge (u, v) (for instance, a path $(\alpha_1, s_1, \omega_1, \alpha_2, s_2, \omega_2)$ will be replaced by $(\alpha_1, s_1, s_2, \omega_2)$).

A compositional workflow specification $G = (V, E)$ together with an authorization policy $A \subseteq S \times U$ will be called a *compositional workflow schema*. An example of a compositional workflow



(a) Ordering on steps



(b) Constraints

s_1	create purchase order
s_2	approve purchase order
s_3	sign GRN
s_3'	sign GRN
s_4	create payment
s_5	countersign GRN
s_6	approve payment
$\neq$	different users must perform steps
$=$	same user must perform steps

(c) Legend

Figure 2: Example of a compositional workflow specification; vertices with no border represent orchestration points

specification is shown in Figure 2. It extends the example in Figure 1 by including orchestration steps and an xor branching. We model the fact that orders below a certain value will not require a countersignature on the GRN. Thus, one branch includes steps to sign and countersign the GRN (which is taken when the value of the order exceeds a certain value), while the other branch contains only the sign GRN step.

2.2 Execution sequences

In a compositional workflow having an xor branching, there exists more than one set of steps that could comprise a workflow instance. And in a compositional workflow having only parallel branching, two different workflow instances will contain the same steps but they may occur in different orders. Here, we introduce the idea of an *execution sequence*, which is an ordered sequence of steps and release points. An execution sequence may be empty. For execution sequences $\sigma = (a_1, \ldots, a_k)$ and $\sigma' = (b_1, \ldots, b_k)$, we define the following two sets of execution sequences:

$$\sigma + \sigma' = \{(a_1, \ldots, a_k, b_1, \ldots, b_\ell)\}$$
$$\sigma * \sigma' = \{(a_1) + \sigma'' : \sigma'' \in (a_2, \ldots, a_k) * (b_1, \ldots, b_\ell)\} \cup \{(b_1) + \sigma'' : \sigma'' \in (a_1, \ldots, a_k) * (b_2, \ldots, b_\ell)\}$$
$$\sigma * () = () * \sigma = \sigma$$

In other words, $\sigma + \sigma'$ represents concatenation of σ and σ'; and $\sigma * \sigma'$ represents all possible interleavings of σ and σ' that preserve the ordering of elements in both σ and σ'. Given sets of execution sequences Σ and Σ', we write $\Sigma + \Sigma'$ to denote $\{\sigma + \sigma' : \sigma \in \Sigma, \sigma' \in \Sigma'\}$ and $\Sigma * \Sigma'$ to denote $\{\sigma * \sigma' : \sigma \in \Sigma, \sigma' \in \Sigma'\}$.

For a compositional workflow G, we write $\Sigma(G)$ to denote the set of execution sequences for G. Then:

- for workflow specification G comprising a single step or release point v, $\Sigma(G) = \{(v)\}$;
- $\Sigma(G_1; G_2) = \Sigma(G_1) + \Sigma(G_2)$;
- $\Sigma(G_1 \parallel G_2) = \Sigma(G_1) * \Sigma(G_2)$; and
- $\Sigma(G_1 \otimes G_2) = \Sigma(G_1) \cup \Sigma(G_2)$.

The possible execution sequences for the example in Figure 2 are:

- $(s_1, s_2, s_4, s_3, s_5, s_6)$
- $(s_1, s_2, s_3, s_4, s_5, s_6)$
- $(s_1, s_2, s_3, s_5, s_4, s_6)$
- $(s_1, s_2, s_4, s_3', s_6)$
- $(s_1, s_2, s_3', s_4, s_6)$

For an execution sequence σ, let σ_S and σ_R be the restriction of σ to the set of steps and release points, respectively. Similarly, let $S(\sigma)$ and $R(\sigma)$ be respectively the set of steps and release points contained in σ.[1]

Given an execution sequence $\sigma = (v_1, \ldots, v_n)$ of G and $i \in [n]$, we define $\text{left}_\sigma(v_i) = (v_1, \ldots, v_{i-1})$, $\text{right}_\sigma(v_i) = (v_{i+1}, \ldots, v_n)$. Also, if $1 \leq i < j \leq n$, then define $\text{btw}_\sigma(v_i, v_j) = (v_{i+1}, \ldots, v_{j-1})$. We will omit the σ subscript from left_σ, right_σ and btw_σ when it is obvious from context.

2.3 Constraints with release points

Suppose we have a requirement that two steps s_1 and s_2 be performed by the same user if a certain instance-specific condition holds; and they should be performed by different users otherwise. In other words, the constraint on the execution on s_1 and s_2 varies depending on the instance.

Release points can be used to encode such requirements by positioning different release points in different, mutually-exclusive branches of the workflow and specifying both constraints on the two steps. Then passing through one branch "switches off" the separation-of-duty constraint, while passing through the other branch switches off the binding-of-duty constraint. In this section, we introduce a formalism for modeling such constraints and their satisfaction.

Let $W = (S \cup R \cup O, E, U, A)$ be a compositional workflow schema. A *constraint with release points* has the form $c = (T, \Theta, P)$, where

[1] Hence, the difference between σ_S and $S(\sigma)$ (resp. σ_R and $R(\sigma)$) is that the former is an ordered sequence, while the latter is a set. In particular, it might be the case, for two ordered sequences σ, σ', that, say, $S(\sigma) = S(\sigma')$ while $\sigma_S \neq \sigma'_S$, in the case where σ and σ' are two different orderings of a same set of steps.

$T \subseteq S$ is the scope of the constraint, $P \subseteq R$ represents the release points of the constraints, and Θ is a family of functions with domain T and range U. For $Q \subseteq S$, we denote by $\Theta|_Q = \{f|_Q : f \in \Theta\}$ the restriction of the family Θ to Q.

Let σ be an execution sequence of W, and $\sigma_P = (r_1, \ldots, r_q)$ be the ordering of release points of P in σ. For every $i \in \{1, \ldots, q-1\}$, define

$$T_0 = T \cap S(\text{left}(r_1));$$
$$T_i = T \cap S(\text{btw}(r_i, r_{i+1})), \text{ for } i \in [q-1];$$
$$T_q = T \cap S(\text{right}(r_q)).$$

In other words, for $i \in [q-1]$, T_i is the set of steps of T occurring between r_i and r_{i+1} in σ.

Given a constraint $c = (T, \Theta, P)$ and an execution sequence σ, we define the *restriction* of c to T_i to be the constraint $c_i = (T_i, \Theta|_{T_i})$. (That is, a constraint with scope limited to T_i and having no release points.) We say that a plan $\pi : S(\sigma) \to U$ *satisfies* c iff for all $i \in \{0, \ldots, q\}$, $\pi|_{T_i}$ satisfies c_i, *i.e.* if $\pi|_{T_i} \in \Theta|_{T_i}$. Informally, a plan satisfies c iff its restriction to each subscope T_i, $i \in \{0, \ldots, q\}$, can be extended to a valid tuple (*i.e.* a tuple which belongs to Θ). We say σ *satisfies* c if there exists a plan $\pi : S(\sigma) \to U$ that satisfies c.

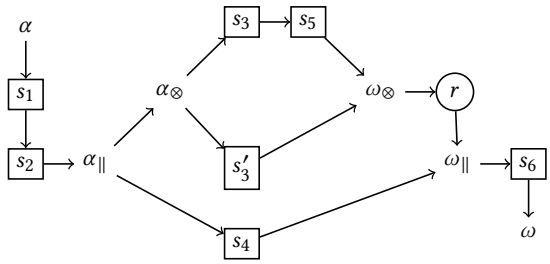
For constraints with a binary scope, such as classical binding-of-duty or separation-of-duty constraints, the addition of release points is a natural generalization. Indeed, a separation-of-duty constraint with two steps s_1, s_2 as scope and P as the set of release points will be satisfied (i) by any plan π if some $r \in P$ occurs between s_1 and s_2, or (ii) by any plan π such that $\pi(s_1) \neq \pi(s_2)$.

For constraints with a larger scope, the meaning of release points is less transparent. Consider, for example, the constraint $\text{atleast}(\{s_1, s_2, s_3, s_4\}, 3, \{r\})$, where r is the release point, and the following assignment:

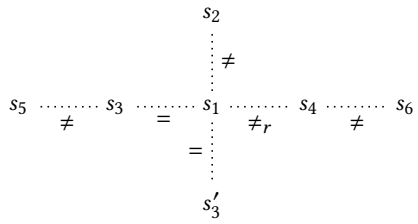
$$\pi(s_1) = \pi(s_3) = u_1, \ \pi(s_2) = \pi(s_4) = u_2.$$

If r occurs before or after all steps in the scope of the constraint in the execution sequence, then this assignment violates the constraint, as only two different users are assigned to these steps. If, however, the execution sequence is (s_1, s_2, s_3, r, s_4), then the constraint is satisfied. Indeed, the restriction of this assignment to $\{s_1, s_2, s_3\}$ can be extended to a valid assignment (by assigning, say, u_3 to s_4). Similarly, the restriction of this assignment to $\{s_4\}$ can also be extended to a valid assignment (assigning, say, u_1 to s_1, u_3 to s_2, and any user to s_3).

We extend our running example by modifying the SoD constraint defined between s_1 and s_4 in order to illustrate how execution sequences and release points might affect the satisfiability of an instance. The resulting workflow specification is illustrated in Figure 3.) Specifically, the constraint is released by r positioned between $\omega_\otimes$ and $\omega_\parallel$. The intuition is to prevent the same person from creating the purchase order and the payment, except when the GRN has been signed (and countersigned, if the upper branch of the xor branching is chosen). Hence, if the "create payment" is processed before the signature/countersignature of the GRN, then the user which has created the purchase order cannot create the payment. Otherwise, if the "create payment" is processed after the signature/countersignature of the GRN, then the SoD constraint is released. In the case where the authorization policy is such that only one user is authorized to execute steps s_1 and s_4, then some execution sequences will be satisfiable, whereas some others will not be satisfiable.



(a) Ordering on steps



(b) Constraints

s_1	create purchase order
s_2	approve purchase order
s_3	sign GRN
s'_3	sign GRN
s_4	create payment
s_5	countersign GRN
s_6	approve payment
r	release point of the constraint blue (s_1, s_4, $\neq$)
$\neq$	different users must perform steps
$=$	same user must perform steps
$\neq_r$	same as $\neq$ but released by r

(c) Legend

Figure 3: A constrained compositional workflow specification with release points; vertices bordered by a rectangle (resp. circle) represent steps (resp. release points); vertices with no border are orchestration points.

Notice that our definitions of constraints with release points allow us to model the SoD and BoD constraints as defined by Basin *et al.* [3]. In their work, a SoD constraint is defined by two sets of steps T_1 and T_2 together with a set of release points P. Then, whenever a user u executes some step $s_1 \in T_1$, this constraint prohibits u from executing any step $s_2 \in T_2$ unless a release point is reached. One can observe that this constraint can be transformed into $|T_1| \cdot |T_2|$ binary SoD constraints with scope $(s_1, s_2) \in T_1 \times T_2$ and release points P. Basin *et al.* define a BoD constraint to be a set of steps T and a set of release points P. Once a user u has executed some step in T, u must execute the remaining steps in T unless a release point is reached. Again, we may transform this into an equivalent set of constraints with binary scope by specifying $\binom{|T|}{2}$ binary BoD constraints with scope $(s, s') \in T \times T$ (with $s \neq s'$) and

release points P. Thus our definition of constraints with release points is considerably more general than existing ones.

A constrained compositional workflow schema (c.c.w.s. for short) is a tuple $(G = (S \cup R \cup O, E), U, A, C)$, where (G, U, A) is a compositional workflow schema, and C is a set of constraints with release points. We assume the scope of a constraint does not contain two exclusive steps. This is a reasonable assumption since two exclusive steps never occur in the same execution sequence. We say constraint $c = (T, \Theta, P)$ is user-independent (UI) iff for every $\theta \in \Theta$ and every permutation $\phi : U \to U$, we have $\phi \circ \theta \in \Theta$.

2.4 WSP with release points

Given a constrained c.c.w.s. $W = (S \cup R \cup O, E, U, A, C)$, we say that an execution sequence σ is *satisfied* if there exists an authorized plan $\pi : S(\sigma) \to U$ that satisfies all constraints in C. Observe that authorization does not depend on the ordering of steps or release points. We say that W is *strongly satisfiable* (resp. *weakly satisfiable*) iff every (resp. at least one) execution sequence of W is satisfiable. We are now able to define the following decision problem which is the main subject of this paper:

WSP with Release Points
Input: A constrained compositional workflow schema $W = (S \cup R \cup O, E, U, A, C)$
Question: Is W strongly satisfiable ?

There is a possibility that in practice weak rather than strong satisfiability is of interest. It is not hard to modify the algorithm described later in the paper to solve the weakly satisfiability version of WSP with Release Points. Clearly WSP with Release Points is a generalization of WSP (indeed, a WSP with Release Points with no xor branching and whose all constraints have no release point is equivalent to a WSP instance), and is thus NP-hard and $W[1]$-hard when parameterized by $k = |S|$ [25]. Moreover, it has been shown that if all considered constraints are user-independent, then WSP can be solved in time $O(2^{k \log_2 k}|W|^{O(1)})$, where k is the number of steps [18] ($|W|^{O(1)}$ means a polynomial in the size of the workflow instance), and that this is the best possible: WSP cannot be solved in time $O(c^{k \log_2 k}|W|^{O(1)})$ for any constant $c < 2$ [15] unless the Strong Exponential Time Hypothesis[2] is false, which is unlikely. This lower bound directly transfers to WSP with Release Points. Despite the seeming difficulty of the problem (since all execution sequences have to be considered), we will be able to show that WSP with Release Points is FPT parameterized by the number of vertices of the DAG (*i.e.* number of steps, release points and orchestration points) if only user-independent constraints are considered.

3 SOLVING THE COMPOSITIONAL WSP WITH RELEASE POINTS

Our aim is thus to provide an algorithm to solve the WSP with Release Points. Recall that the problem asks whether *every* execution sequence is satisfiable. Hence, a naive approach would be to enumerate all execution sequences, and test whether each of them is satisfiable. In the next section, we show that such an exhaustive enumeration is wasteful. More precisely, we define an equivalence relation over execution sequences, and show that all execution sequences which belong to the same equivalence class behave the same with respect to satisfiability.

[2]The Strong Exponential Time Hypothesis [16] states that a CNF SAT formula on n variables cannot be solved in c^n time for any $c < 2$.

3.1 Execution arrangements

Let $\sim$ be the following relation over the set of all execution sequences of a workflow: $\sigma \sim \sigma'$ iff (i) $\sigma_R = \sigma'_R$ (ii) $S(\sigma) = S(\sigma')$ and (iii) for all $s \in S$, $R(\text{right}_\sigma(s)) = R(\text{right}_{\sigma'}(s))$. It is easy to see that $\sim$ defines an equivalence relation. Its equivalence classes are called *execution arrangements*. Informally, all execution sequences of an execution arrangement have the same set of steps and release points, their release points are in the same order, and every step occurs between the same pair of release points.

From this observation, it makes sense to define a "compact" representation of an execution arrangement. More precisely, we define an execution arrangement as an ordered sequence $(S_1, r_1, S_2, r_2, ..., r_{q-1}, S_q)$ which satisfies the following properties:

(1) $\{S_1, ..., S_q\}$ is a partition of S (we may have $S_i = \emptyset$ for some $i \in [q]$);
(2) $(r_1, ..., r_{q-1})$ is a linear subextension of G containing all release points;
(3) for all $(s_1, \ldots, s_q) \in S_1 \times \cdots \times S_q$, $(s_1, r_1, \ldots, r_{q-1}, s_q)$ is a linear subextension of G.

Notice the abuse of notation in the last property if $S_i = \emptyset$ for some $i \in [q]$. In this case, we simply omit such steps s_i in the sequence $(s_1, r_1, \ldots, r_{q-1}, s_q)$. For instance, if $S_2 = \emptyset$, then the sequence is actually $(s_1, r_1, r_2, s_3, \ldots, r_{q-1}, s_q)$.

The execution arrangements and the corresponding execution sequences for the example in Figure 3 are tabulated below.

Arrangement	Sequence
$\{s_1, s_2, s_3, s_5\}, r, \{s_4, s_6\}$	$(s_1, s_2, s_3, s_5, r, s_4, s_6)$
$\{s_1, s_2, s_3, s_4, s_5\}, r, \{s_6\}$	$(s_1, s_2, s_3, s_5, s_4, r, s_6)$ $(s_1, s_2, s_3, s_4, s_5, r, s_6)$ $(s_1, s_2, s_4, s_3, s_5, r, s_6)$
$\{s_1, s_2, s'_3\}, r, \{s_4, s_6\}$	$(s_1, s_2, s'_3, r, s_4, s_6)$
$\{s_1, s_2, s'_3, s_4\}, r, \{s_6\}$	$(s_1, s_2, s'_3, s_4, r, s_6)$ $(s_1, s_2, s_4, s'_3, r, s_6)$

As we can see, even with one xor branching and one release point, the number of execution arrangements (4) is smaller than the number of execution sequences (7). Naturally, this difference increases with the number of xor branchings and release points.

The idea of defining this equivalence relation comes from the fact that the ordering of steps between two release points is of no importance for determining the satisfiability of a given execution sequence. We will exploit this property and prove that the satisfiability of two execution sequences of an execution arrangement are equivalent, *i.e.* one is satisfiable iff the other is. This is formalized by the following lemma.

Lemma 3.1. *Let $W = (G = (S \cup R \cup O, E), U, A, C)$ be a c.c.w.s.. Given two execution sequences σ, σ' of W with $\sigma \sim \sigma'$, σ is satisfiable if and only if σ' is.*

Proof. Let $c = (T, \Theta, R) \in C$. By definition of $\sim$, we have $\sigma_R = \sigma'_R = (r_1, \ldots, r_q)$. Now, let $i \in \{1, \ldots, q-1\}$, and denote by T_i the set

$T \cap S(\text{btw}_\sigma(r_i, r_{i+1}))$ and by T'_i the set $T \cap S(\text{btw}_{\sigma'}(r_i, r_{i+1}))$. Again by definition of $\sim$, it holds that $R(\text{right}_\sigma(s)) = R(\text{right}_{\sigma'}(s))$ for every $s \in S(\sigma)$, which implies $S(\text{btw}_\sigma(r_i, r_{i+1})) = S(\text{btw}_{\sigma'}(r_i, r_{i+1}))$, and thus $T_i = T'_i$. It proves that σ satisfies c iff σ' satisfies c. Finally, recall that authorization does not depend on the ordering of steps or release points. Hence, since $S(\sigma) = S(\sigma')$ by definition, an authorized plan for σ will also be an authorized plan for σ', and conversely. □

Lemma 3.1 states that in order to test the satisfiability of a c.c.w.s., it is sufficient to test the satisfiability of only one execution sequence per execution arrangement. Observe that the number of possible execution sequences can be as large as $(|S| + |R|)!$, even with no xor branching, while the number of execution arrangements is bounded above by $|R|!|R|^{|S|}$.

Thus, the main issue is now to enumerate all possible execution arrangements of an instance, and, for each of them, to test its satisfiability. The enumeration is itself a non-trivial question, not least because of the possible interleaving of several xor and parallel branchings. In particular, the presence of xor branchings implies that the set of steps and release points might be different depending on the executions. Hence, our approach can be decomposed into three subtasks:

(1) elimination of xor branchings;
(2) enumeration of all execution arrangements of a xor-free instance; and
(3) testing the satisfiability of an execution arrangement.

The next three subsections address these subtasks in turn. In Section 3.2, we develop a method for decomposing a problem instance into subproblems that do not contain any xor branching. Our algorithm will run in FPT time parameterized by the number of xor branchings of the instance, and polynomial space. In Section 3.3, we describe an algorithm to enumerate execution arrangements running in FPT time parameterized by the number of steps and release points, and using polynomial space. Finally, in Section 3.4, we show that each subproblem can be reduced to the classical WSP, allowing us to use any known method for solving this problem in order to terminate the algorithm.

Algorithm 1 summarizes the general procedure in an informal manner. In Section 3.5, we provide a theoretical analysis of our algorithm.

Algorithm 1 General algorithm

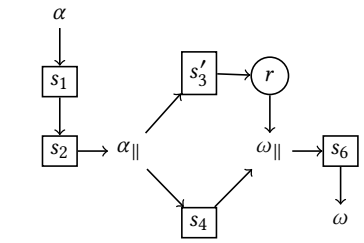
Input: $W = (S, R, O, E, U, A, C)$ a c.c.w.s.
1: **for all** xor-free subinstance W' **do**
2: **for all** execution arrangement Σ of W' **do**
3: **if** Σ is unsatisfiable **then**
4: **return** UNSATISFIABLE INSTANCE
5: **end if**
6: **end for**
7: **end for**
8: **return** SATISFIABLE INSTANCE

3.2 Elimination of xor branchings

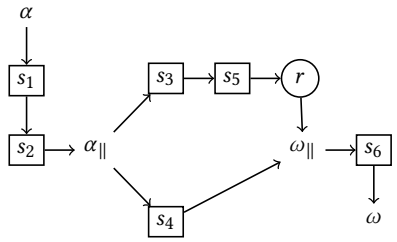
Recall that in an execution sequence σ of a compositional workflow specification containing a xor branching of two subworkflows $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, either $V_1 \subseteq V(\sigma)$ or $V_2 \subseteq V(\sigma)$. Such a branching is identified by its corresponding input and output vertices $\alpha_\otimes$ and $\omega_\otimes$, respectively. For such a pair $x = (\alpha_\otimes, \omega_\otimes)$, we construct the compositional workflow schemas G_1^x and G_2^x from G by removing all vertices from G_2 and G_1, respectively. Now, given a set X of pairs of *xor* input and output vertices, we define the set of *reduced compositional workflows* as follows:

- if $X = \{x\}$, then $red_X(G) = \{G_1^x, G_2^x\}$;
- otherwise, for an arbitrary $x \in X$ whose branches do not contain themselves another xor branching, $red_X(G) = red_{X\setminus\{x\}}(G_1^x) \cup red_{X\setminus\{x\}}(G_2^x)$.

Figure 4 illustrates these definitions applied to our running example. In the first workflow, steps s_3 and s_5 are removed, while in the second one, step s'_3 is removed (then both are simplified using rules described in Section 2.1, allowing us to remove $\alpha_\otimes$ and $\omega_\otimes$).



(a) first xor-free workflow



(b) second xor-free workflow

Figure 4: The two workflows obtained after removing the xor branching of Figure 3.

We denote by $\mathcal{B}$ be the set of all pairs of xor input and output vertices of a given c.c.w.s. $W = (G = (S \cup R \cup O, E), U, A, C)$. Informally, $red_\mathcal{B}(G)$ contains all possible compositional workflows obtained from G by removing, for every xor branching, one of the two branches. Hence, any $G' \in red_\mathcal{B}(G)$ is xor-free, and, in particular, does not contain two exclusive steps. For $G' = (V', E') \in red_\mathcal{B}(G)$, let $W[G']$ be the c.c.w.s. induced by G': $W[G'] = (G' = (S \cap V', R \cap V', O \cap V', E), U, A \cap (V' \times U), C)$ (as mentioned earlier, we may assume that constraints do not contain exclusive steps, hence there is no need for restricting the scopes of constraints). We now use this construction in the following result.

Lemma 3.2. *Using the notation above, W is satisfiable if and only if $W[G']$ is satisfiable for every $G' \in red_\mathcal{B}(G)$.*

Proof. Simply observe that every execution sequence of $W[G']$ is also an execution sequence of W, and, conversely, for every execution sequence σ of W, there exists $G' \in red_{\mathcal{B}}(G)$ such that σ is an execution sequence of $W[red_{\mathcal{B}}(G)]$. □

3.3 Enumeration of execution arrangements

Throughout this subsection, we will assume we are given a c.c.w.s. $W = (G = (S, R, O, E), U, A, C)$ which does not contain any xor branching. Our objective is to provide an algorithm enumerating all execution arrangements of W.

Since W is assumed to be xor-free, we know that all execution sequences (and thus all execution arrangements) that can be obtained from G have the same set of steps and release points, namely S and R, respectively.

Let us recall the properties satisfied by an execution arrangement $(S_1, r_1, S_2, r_2, ..., r_{q-1}, S_q)$:

(1) $\{S_1, ..., S_q\}$ is a partition of S (we may have $S_i = \emptyset$ for some $i \in [q]$);
(2) $(r_1, ..., r_{q-1})$ is a linear subextension of G containing all release points;
(3) for all $(s_1, \ldots, s_q) \in S_1 \times \cdots \times S_q$, $(s_1, r_1, \ldots, r_{q-1}, s_q)$ is a linear subextension of G.

The first step of the algorithm is to enumerate all linear subextensions of release points. This is actually equivalent to the enumeration of all topological orderings of the partial order restricted to R, and can be done using a BFS-based recursive algorithm (although more involved and efficient algorithms exist, see *e.g.* [17, 20, 22, 24]). Hence, in the following, we fix such a linear extension[3] $(r_1, \ldots, r_{q-1})$.

For the sake of readability, we will now restrict ourselves to steps only: we first assume that G does not contain release points, by considering the restriction of the partial order to $V \setminus R$. In addition, we will consider orchestration points as normal steps. In order to obtain execution arrangements containing "concrete" steps only, simply remove the orchestration points once an execution arrangement is returned. Hence, we now assume $V = S$.

Our procedure is described in detail in Algorithm 2, and consists in constructing the partition $S_1, \ldots S_q$ step by step. Also, it takes as input a partial partition $S_1, \ldots, S_q$ of a subset of S. It takes as input the subset $S_{rem} \subseteq S$ of remaining steps that have to be assigned to some set of the partition $\{S_1, \ldots, S_q\}$ of $S \setminus S_{rem}$. For the first call, simply set $S_{rem} = S$ and $S_i = \emptyset$ for all $i \in [q]$.

Once a step s has been chosen (line 4), we need to decide to which set it can belong to. To do so, we determine two indices i_{min} and i_{max} such that for all $i \in \{i_{min}, \ldots, i_{max}\}$, s can be put in S_i. Roughly speaking, we cannot put s to the left of a set S_j such that $s' < s$ for some $s' \in S_j$, or to the left of some release point r_j such that $r_j < s$ (and, similarly, to the right of a set S_j or a release point r_j such that $s < r_j$ or $s < s'$ for some $s' \in S_j$). This is illustrated in Figure 5.

Lemma 3.3. *Every output of Algorithm 2 is an execution arrangement, and every execution arrangement is an output of Algorithm 2.*

[3]We could incorporate the enumeration of sequences of release points inside Algorithm 2. However, for the sake of readability, we choose to separate this step.

Algorithm 2 Enumeration of execution arrangements given a linear extension $(r_1, \ldots, r_{q-1})$ of release points

Input: $S_{rem} \subseteq S$, $\{S_1, \ldots, S_q\}$ partition of $S \setminus S_{rem}$
1: **if** $S_{rem} = \emptyset$ **then**
2: output $(S_1, r_1, S_2, r_2, \ldots, r_{q-1}, S_q)$
3: **else**
4: $s \leftarrow$ source of $G[S_{rem}]$ (arbitrarily chosen)
5: $i_{min} \leftarrow \max(\{i \in \{2, \ldots, q\} : r_{i-1} < s$ or $s' < s$ for some $s' \in S_i\} \cup \{1\})$
6: $i_{max} \leftarrow \min(\{i \in \{1, \ldots, q-1\} : s < r_i$ or $s < s'$ for some $s' \in S_i\} \cup \{q\})$
7: **for all** $i \in \{i_{min}, \ldots, i_{max}\}$ **do**
8: make a recursive call with input $S_{rem} \setminus \{s\}$, $\{S_1, \ldots, S_i \cup \{s\}, \ldots, S_q\}$
9: **end for**
10: **end if**

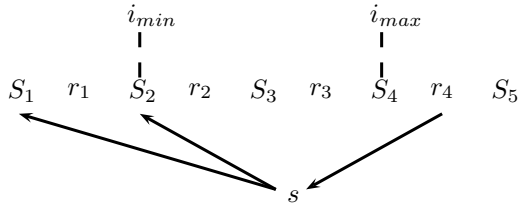


Figure 5: Illustration of i_{max} and i_{min}. Arrows indicate that $s < r_4$ and that there exists $s'_1 \in S_1$ and $s'_2 \in S_2$ such that $s'_1 < s$ and $s'_2 < s$. Hence, s may belong to S_2, S_3 or S_4.

Proof. Let $\Sigma = (S^*_1, r_1, S^*_2, \ldots, r_{q-1}, S^*_q)$ be an output of our algorithm, and let us show it is indeed an execution arrangement. First, $(r_1, \ldots, r_{q-1})$ is a fixed linear extension of R, and the algorithm only stops when all steps have been assigned to a set S_i, thus properties (2) and (1) are obviously satisfied. For all inputs $(S_1, \ldots, S_q)$ of the algorithm, we prove that for all $(s_1, \ldots, s_q) \in S_1 \times \cdots \times S_q$, $(s_1, r_1, \ldots, r_{q-1}, s_q)$ satisfies property 3, by induction on $|\bigcup_{i \in [q]} S_i|$. The property is obviously true at the first call since $(r_1, \ldots, r_{q-1})$ is a linear subextension of R. Then, let $(S_1, \ldots, S_q)$ be an input satisfying the property, and s chosen at line 4. Let $i \in \{i_{min}, \ldots, i_{max}\}$. For all $j < i$, by definition of i_{min}, it holds that $s \not< r_j$, and $s \not< s'$ for all $s' \in S_j$. Similarly, for all $j \geq i$ we have $r_j \not< s$, and, for all $j > i$ and all $s' \in S_j$, we have $s' \not< s$. This proves that $(s_1, r_1, \ldots, r_{q-1}, s_q)$ is a linear subextension of G for all $(s_1, \ldots, s_q) \in S_1 \times \cdots \times S_i \cup \{s\} \times \cdots \times S_q$.

Conversely, let $\Sigma = (S^*_1, r_1, \ldots, r_{q-1}, S^*_q)$ be an execution arrangement. We now show that Σ is an output of the algorithm. To do so, assume that there is a call of the algorithm with input $\{S_1, \ldots, S_q\}$ such that $S_i \subseteq S^*_i$ for all $i \in [q]$ (this is obviously true at the first call). Let s be the step chosen at line 4, and i^* such that $s \in S^*_i$. We need to show that $i_{min} \leq i^* \leq i_{max}$. If $i_{min} = 1$ then the first inequality obviously holds. Otherwise, the definition of i_{min} implies that we have $r_{i_{min}-1} < s$ or $s' < s$ for some $s' \in S_{i_{min}}$. In both cases, having $i^* < i_{min}$ would break property 3 and Σ would not be an execution arrangement. Similarly, the second inequality holds

trivially if $i_{max} = q$, and $i_{max} < i^*$ together with the definition of i_{max} would imply that Σ is not an execution arrangement. □

3.4 Reduction to WSP

It now remains to test the satisfiability of an execution arrangement. Indeed, as we saw in Lemma 3.1, all execution sequences of the same execution arrangement behave the same with respect to satisfiability. To do so, we show that satisfiability of an execution arrangement reduces to the satisfiability of a finite number of "classical" WSP instances. We recall the formal definition of WSP [25].

> Workflow Satisfiability Problem (WSP)
> *Input:* A constrained workflow authorization schema $W = (G = (S, E), U, A, C)$
> *Question:* Is there a valid plan $\pi : S \to U$?

Let $\Sigma = (S_1, r_1, S_2, r_2, ..., r_{q-1}, S_q)$ be an execution arrangement (*i.e.* an output of Algorithm, line 2), and $c = (T, \Theta, P)$ be a constraint with release points $P = \{r_{p_1}, \ldots, r_{p_{|P|}}\}$ (*w.l.o.g.* we assume $p_i \leq p_j$ whenever $i \leq j$, *i.e.* this ordering is a linear extension of $R(\Sigma)$). As in Section 2.3, for all $i \in \{1, \ldots, |P| - 1\}$, define $T_i = T \cap S(\text{btw}(r_{p_i}, r_{p_{i+1}}))$, $T_0 = T \cap S(\text{left}(r_{p_0}))$, $T_{|P|} = T \cap S(\text{right}(r_{p_{|P|}}))$, and the "classical" constraint $c_i = (T_i, \Theta|_{T_i})$. Recall that c is satisfied by an execution sequence σ iff there exists a plan π such that $\pi|_{T_i}$ satisfies c_i for every $i \in \{0, \ldots, q\}$. Thus, for each $i \in [|P|]$, it makes sense to define the WSP instance $W_i = (G_i = (S_i, E_i), U, A_i, C_i)$, which defines the partial order of G restricted to S_i, $A_i = A \cap (S_i \times U)$ and $C_i = \{c_i | c \in C\}$. By the foregoing, we obtain the following result:

Lemma 3.4. *Σ is satisfiable (for WSP with Release Points) if and only if W_i is satisfiable (for WSP) for every $i \in [|P|]$.*

Using this result, we are thus able to use any state-of-the-art solver for WSP as a black box in order to obtain the general algorithm. There are several papers describing the design and evaluation of practical WSP algorithms, see, e.g., [7, 18, 19, 25]. Some of these algorithms are bespoke, while others use SAT solvers.

3.5 Analysis of the algorithm

We now analyze the running time and space of our algorithm with respect to the different parameters of an instance: the number of users $|U|$, the number of constraints $|C|$, the number of steps $|S|$, the number of release points $|R|$ and the number of orchestration points $|O|$. We denote by $|W|$ the total size of an instance. First, observe that we always have $|O| = O(|R| + |S|)$ by construction (in practice, $|O|$ will be much smaller than $|S| + |R|$ because of the simplification mentioned at the end of Section 2.1). We will also consider the number $|\mathcal{B}|$ of xor branchings in a problem instance. (Clearly $|\mathcal{B}| \leq |O|$.)

Most techniques we use in this algorithm are based on *recursive procedures*. Given an input I, such a recursive procedure applies various operations (dependent on I), and then makes one or several recursive calls with inputs $I_1, \ldots, I_w$. In order for such a procedure to terminate, there must exist an integer-valued *measure* $\ell(I)$ which strictly decreases with each recursive call, *i.e.* such that $\ell(I_j) < \ell(I)$ for all $j \in [w]$. For instance, the measure of the recursive procedure of Section 3.2 is the number of xor branchings in the input, which decreases by one at each new call, while the measure of Algorithm 2 is the number of steps, which also decreases by one at each new call.

The *width* of a recursive algorithm is the maximum number of recursive calls at each step (*i.e.* w in the previous notation), while the *depth* is the measure $\ell(I)$ of the first input of the algorithm. Then a recursive algorithm has a running time of $O(w^{\ell(I)}T(I))$, where $T(I)$ is the running time of a single call, and a space complexity of $O(\ell(I)Sp(I))$, where $Sp(I)$ is the space complexity of a single call.

The worst case complexity (time or space) of our algorithm is the product of the respective complexity of the algorithms for solving the three subproblems:

(1) enumeration of all xor-free subinstances;
(2) given a xor-free instance, enumeration of all execution arrangements;
(3) given an execution arrangement, reduction to WSP and satisfiability test.

The first step, described in Section 3.2, uses a recursive algorithm. Its branching width is 2, its depth is $|\mathcal{B}|$ (the number of xor-branchings of the instance), and every step takes polynomial time and space, since it simply consists in removing some vertices of the workflow specification. Thus, the algorithm uses polynomial space and its running time is $O(2^{|\mathcal{B}|} \cdot |W|^{O(1)})$.

Given a xor-free instance of the previous step, the next task is to enumerate all execution arrangements (Section 3.3). To do so, we first enumerate all linear extensions of release points. This can be done in time linear in the number of such linear extensions [22], which is at most $|R|!$. Then, given a linear extension of the release points, we can apply Algorithm 2, which is a recursive algorithm, whose branching width is at most q, the number of release points plus one (see line 7), and depth is $|S|$ (since we remove an element of S_{rem} at each recursive call). Moreover, each call takes polynomial time and space. Hence Algorithm 2 takes time $O(q^{|S|} \cdot |W|^{O(1)})$. Thus subproblem 2 uses polynomial space and its running time is

$$O(|R|!q^{|S|}|W|^{O(1)}) = O(|R|!(|R| + 1)^{|S|}|W|^{O(1)}).$$

Finally, the last step contains a reduction to several instances of WSP, and a satisfiability test for each of them. More precisely, given an execution arrangement Σ, we construct, in polynomial time, $|R(\Sigma)| + 1 = O(|R|)$ instances of WSP. Then, the running-time of the satisfiability test of each WSP instance depends on the chosen algorithm. Let $wsp(\alpha, \beta, \gamma)$ be the running time of an algorithm solving a WSP instance with α users, β steps and γ constraints. The running time of this step is thus $O(|R|)wsp(|U|, |S|, |C|)$, while the space complexity is the one of the chosen algorithm for WSP. If all constraints are user-independent, then the algorithm of [18] runs in time $wsp(|U|, |S|, |C|) = O(2^{|S|\log_2|S|}|W|^{O(1)})$ and polynomial space. Thus, this step takes time $O(2^{|S|\log_2|S|}|W|^{O(1)})$ and polynomial space.

In total, the running-time of our algorithm is thus

$$O(2^{|\mathcal{B}|}|R|!(|R| + 1)^{|S|}wsp(|U|, |S|, |C|)|W|^{O(1)}).$$

If all constraints are user-independent, this becomes

$$O(2^{|\mathcal{B}|}|R|!(|R| + 1)^{|S|}2^{|S|\log_2|S|}|W|^{O(1)}),$$

which is an FPT running time parameterized by the number of vertices of the workflow specification. Moreover, the algorithm uses polynomial space. Thus, we obtain the following result.

Theorem 3.5. *If all constraints are user-independent, WSP with Release Points can be solved in time*

$$O(2^{|\mathcal{B}|}|R|!(|R|+1)^{|S|}2^{|S|\log_2(|S|)}|W|^{O(1)})$$

and polynomial space.

As long as the values of $|\mathcal{B}|$ and $|R|$ are small, our algorithm may well be efficient in practice since the branch-and-bound algorithm of [18] has proved to be very efficient in practice and was further improved in [19]. For instance, in the particular case where $|R| = 0$, we obtain the same running time as for WSP. Observe that this algorithm scales polynomially with the number of users which is likely to be, in practice, the largest parameter of a workflow. Finally, our algorithm is deterministic, *i.e.* does not produce false positive or false negative answers, contrary to the algorithm of Basin *et al.* [3]. It also uses polynomial space, contrary to the algorithm of Crampton and Gutin [10].

4 RELATED WORK

Research on workflow satisfiability began with the seminal work of Bertino, Ferrari and Atluri [4] and Crampton [8]. Wang and Li were the first to demonstrate that WSP, subject to specific and limiting restrictions, was fixed-parameter tractable [25]. A substantial body of work now exists on the fixed-parameter tractability of WSP [6, 9, 11]. In particular, it is known that WSP is fixed-parameter tractable (parameterized by the number of steps) when all constraints are regular [11] or user-independent [6].

Basin, Burri and Karjoth introduced the notion of release points [3] in order to model workflows in which the set of steps that are executed may vary and for which constraints only apply to certain sets of steps. They modeled workflows using a process algebra and define the notion of an *enforcement process*, which corresponds to a valid plan in our model of workflow satisfiability. They showed that the enforcement process existence (EPE) problem, which corresponds to the workflow satisfiability problem, is NP-hard, and developed a polynomial-time heuristic to solve the EPE problem. Their algorithm achieves good results under the assumption that the user population is large and "the static authorizations are equally distributed between them".

We believe it is reasonable to assume the user population is large, at least relative to the number of steps in the workflow. Indeed, our FPT algorithms are of interest provided this assumption holds. However, it is unclear whether it is reasonable to assume that static authorizations are equally distributed. We adopt a different approach by extending an existing model for compositional workflows, due to Crampton and Gutin [10], to accommodate release points, and modifying the definition of constraint satisfaction and workflow satisfiability accordingly. By making use of existing work on WSP we are able to provide the first FPT algorithm for WSP with release points. Moreover, this algorithm is exact and may be used for any workflow specification containing user-independent constraints. This is in contrast to the work by Basin *et al.*, which yields a non exact algorithm, in the sense that it may produce false negatives (although it does run in polynomial time) and only applies to specific SoD and BoD constraints. However, it should also be noted that the approach of Basin *et al.* can model more complex workflow specifications, such as ones containing loops. In other words, their approach is applicable to more workflow patterns than ours, but to fewer types of workflow constraints.

On the other hand, there exists work on workflow satisfiability with more complex control flow patterns that does not consider release points [5, 10, 27], of which only the work of Crampton and Gutin [10] considers fixed-parameter tractability of WSP. One contribution of this paper is to extend the model due to Crampton and Gutin [10], but we also introduce the notion of execution arrangements and an algorithm which considers execution arrangements (rather than execution sequences). Thus we provide techniques that can usefully be applied to WSP for compositional workflows without release points.

5 CONCLUDING REMARKS

In this paper, we have extended recent work on FPT algorithms for the workflow satisfiability problem by introducing release points. Release points allow constraints to be defined for a workflow specification in which the set of steps that is executed may vary from one workflow instance to another. In particular, a constraint can be "switched on" when certain steps are executed in a certain sequence and "switched off" otherwise. The typical use case is when there is non-deterministic branching in the specification and the constraint should apply when one branch is executed but not the other. As such, this work allows us to further close the gap between the workflow specifications that are required in practice and those for which we can provide algorithms to solve the workflow satisfiability problem. In particular, our algorithms can be used as the basis for an on-line reference monitor for workflows containing xor branching (applying methods described by Crampton and Gutin [10, Section 2.2]).

We plan to extend our model to include sub-workflows that can be repeated. A purchase order workflow, for example, might include a sub-workflow containing a single step that creates an item in a purchase order. We expect that some care will be required to integrate looping constructs and release points.

In Section 3.5 we noted that we reduce WSP with Release Points to WSP and use existing WSP solvers. The performance of such solvers has improved dramatically in recent years [7, 18, 19, 25]. We plan to use state-of-the-art solvers to test the hypothesis that strong satisfiability for real-world workflow specifications with xor branching and release points can be solved efficiently in practice.

It may also be interesting to consider the workflow satisfiability problem when the authorization policy changes over the lifetime of a workflow instance. Such changes might occur, for example, if some users are unavailable at certain times. Some related prior work exists on workflow resiliency [21, 25]. It is also possible to model certain constraints with release points by modifying the authorization policy. Indeed, this is essentially how Basin *et al.* define enforcement processes for their SoD and BoD constraints [3].

Finally, recent work has shown that WSP is FPT for class-independent constraints [9], a generalization of user-independent constraints that allow for the specification of constraints over

groups of users. Such constraints are useful for specifying requirements determined by organizational structures. It would be interesting to investigate whether WSP with release points remains FPT when we allow class-independent constraints in the workflow specification.

Aknowledgement. Gregory Gutin's research was supported by Royal Society Wolfson Research Merit Award.

REFERENCES

[1] American National Standards Institute. *ANSI INCITS 359-2004 for Role Based Access Control*, 2004.
[2] Bang-Jensen, J., and Gutin, G. *Digraphs - theory, algorithms and applications*. Springer, 2002.
[3] Basin, D. A., Burri, S. J., and Karjoth, G. Obstruction-free authorization enforcement: Aligning security and business objectives. *Journal of Computer Security 22*, 5 (2014), 661–698.
[4] Bertino, E., Ferrari, E., and Atluri, V. The specification and enforcement of authorization constraints in workflow management systems. *ACM Trans. Inf. Syst. Secur. 2*, 1 (1999), 65–104.
[5] Bertolissi, C., Santos, D. R. D., and Ranise, S. Automated synthesis of run-time monitors to enforce authorization policies in business processes. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015* (2015), F. Bao, S. Miller, J. Zhou, and G. Ahn, Eds., ACM, pp. 297–308.
[6] Cohen, D., Crampton, J., Gagarin, A., Gutin, G., and Jones, M. Iterative plan construction for the workflow satisfiability problem. *J. Artif. Intell. Res. (JAIR) 51* (2014), 555–577.
[7] Cohen, D. A., Crampton, J., Gagarin, A., Gutin, G., and Jones, M. Algorithms for the workflow satisfiability problem engineered for counting constraints. *J. Comb. Optim. 32*, 1 (2016), 3–24.
[8] Crampton, J. A reference monitor for workflow systems with constrained task execution. In *10th ACM Symposium on Access Control Models and Technologies, SACMAT 2005, Stockholm, Sweden, June 1-3, 2005, Proceedings* (2005), E. Ferrari and G. Ahn, Eds., ACM, pp. 38–47.
[9] Crampton, J., Gagarin, A., Gutin, G., Jones, M., and Wahlström, M. On the workflow satisfiability problem with class-independent constraints for hierarchical organizations. *ACM Trans. Priv. Secur. 19*, 3 (2016), 8:1–8:29.
[10] Crampton, J., and Gutin, G. Constraint expressions and workflow satisfiability. In *18th ACM Symposium on Access Control Models and Technologies, SACMAT '13, Amsterdam, The Netherlands, June 12-14, 2013* (2013), M. Conti, J. Vaidya, and A. Schaad, Eds., ACM, pp. 73–84.
[11] Crampton, J., Gutin, G., and Yeo, A. On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Trans. Inf. Syst. Secur. 16*, 1 (2013), 4:1–4:31.
[12] Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. *Parameterized Algorithms*. Springer, 2015.
[13] Diestel, R. *Graph Theory, 4th Edition*, vol. 173 of *Graduate texts in mathematics*. Springer, 2012.
[14] Downey, R. G., and Fellows, M. R. *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer, 2013.
[15] Gutin, G., and Wahlström, M. Tight lower bounds for the workflow satisfiability problem based on the strong exponential time hypothesis. *Inf. Process. Lett. 116*, 3 (2016), 223–226.
[16] Impagliazzo, R., and Paturi, R. On the complexity of k-SAT. *J. Comput. Syst. Sci. 62*, 2 (2001), 367–375.
[17] Kalvin, A. D., and Varol, Y. L. On the generation of all topological sortings. *Journal of Algorithms 4*, 2 (1983), 150 – 162.
[18] Karapetyan, D., Gagarin, A. V., and Gutin, G. Pattern backtracking algorithm for the workflow satisfiability problem with user-independent constraints. In *Frontiers in Algorithmics - 9th International Workshop, FAW 2015, Guilin, China, July 3-5, 2015, Proceedings* (2015), J. Wang and C. Yap, Eds., vol. 9130 of *Lecture Notes in Computer Science*, Springer, pp. 138–149.
[19] Karapetyan, D., Parkes, A. J., Gutin, G., and Gagarin, A. Pattern-based approach to the workflow satisfiability problem with user-independent constraints. *CoRR abs/1604.05636* (2016).
[20] Knuth, D. E., and Szwarcfiter, J. L. A structured program to generate all topological sorting arrangements. *Inf. Process. Lett. 2*, 6 (1974), 153–157.
[21] Mace, J. C., Morisset, C., and van Moorsel, A. P. A. Quantitative workflow resiliency. In *Computer Security - ESORICS 2014 - 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part I* (2014), M. Kutylowski and J. Vaidya, Eds., vol. 8712 of *Lecture Notes in Computer Science*, Springer, pp. 344–361.
[22] Pruesse, G., and Ruskey, F. Generating linear extensions fast. *SIAM J. Comput. 23*, 2 (1994), 373–386.
[23] van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., and Barros, A. P. Workflow patterns. *Distributed and Parallel Databases 14*, 1 (2003), 5–51.
[24] Varol, Y. L., and Rotem, D. An algorithm to generate all topological sorting arrangements. *Comput. J. 24*, 1 (1981), 83–84.
[25] Wang, Q., and Li, N. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur. 13*, 4 (2010), 40.
[26] White, S., and Miers, D. *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Future Strategies Incorporated, 2008.
[27] Yang, P., Xie, X., Ray, I., and Lu, S. Satisfiability analysis of workflows with control-flow patterns and authorization constraints. *IEEE Trans. Services Computing 7*, 2 (2014), 237–251.