On the Workflow Satisfiability Problem with Class-independent Constraints

Jason Crampton, Andrei Gagarin, Gregory Gutin, and Mark Jones

Royal Holloway, University of London, Egham, Surrey TW20 0EX, UK

Abstract

A workflow specification defines sets of steps and users. An authorization policy determines for each user a subset of steps the user is allowed to perform. Other security requirements, such as separation-of-duty, impose constraints on which subsets of users may perform certain subsets of steps. The *workflow satisfiability problem* (WSP) is the problem of determining whether there exists an assignment of users to workflow steps that satisfies all such authorizations and constraints. An algorithm for solving WSP is important, both as a static analysis tool for workflow specifications, and for the construction of run-time reference monitors for workflow management systems. Given the computational difficulty of WSP, it is important, particularly for the second application, that such algorithms are as efficient as possible.

We introduce class-independent constraints, enabling us to model scenarios where the set of users is partitioned into groups, and the identities of the user groups are irrelevant to the satisfaction of the constraint. We prove that solving WSP is fixed-parameter tractable (FPT) for this class of constraints and develop an FPT algorithm that is useful in practice. We compare the performance of the FPT algorithm with that of SAT4J (a pseudo-Boolean SAT solver) in computational experiments, which show that our algorithm significantly outperforms SAT4J for many instances of WSP. User-independent constraints, a large class of constraints including many practical ones, are a special case of class-independent constraints for which WSP was proved to be FPT (Cohen *et al.*, J. Artif. Intel. Res. 2014). Thus our results considerably extend our knowledge of the fixed-parameter tractability of WSP.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Workflow Satisfiability Problem, Constraint Satisfaction Problem, fixed-parameter tractability, user-independent constraints

Digital Object Identifier 10.4230/LIPIcs.IPEC.2015.66

1 Introduction

It is increasingly common for organizations to computerize their business and management processes. The co-ordination of the tasks or steps that comprise a computerized business process is managed by a workflow management system (or business process management system). Typically, the execution of these steps will be triggered by a human user, or a software agent acting under the control of a human user, and each step may only be executed by an *authorized* user. Thus a workflow specification will include an authorization policy defining which users are authorized to perform which steps.

In addition, many workflows require controls on the users that perform certain sets of steps [1, 2, 3, 7, 14]. Consider a simple purchase-order system in which there are four steps: raise-order (s_1), acknowledge-receipt-of-goods (s_2), raise-invoice (s_3), and send-payment (s_4). The workflow specification for the purchase-order system includes rules to prevent fraudulent use of the system, the rules taking the form of *constraints* on users that can perform pairs of steps in the workflow: the same user may not raise the invoice (s_3) and sign for the

10th International Symposium on Parameterized and Exact Computation (IPEC 2015).
Editors: Thore Husfeldt and Iyad Kanj; pp. 66–77

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

goods (s_2), for example. Such a constraint is known as a *user-independent* (UI) constraint, since the specific identities of the users that perform these steps are not important, only the relationship between them (in this example, the identities must be different).

Once we introduce constraints on the execution of workflow steps, it may be impossible to find a *valid plan* – an assignment of authorized users to workflow steps such that all constraints are satisfied. The Workflow Satisfiability Problem (WSP) takes a workflow specification as input and outputs a valid plan if one exists. WSP is known to be NP-hard, even when the set of constraints only includes constraints having a relatively simple structure (and arising regularly in practice). In particular, the Graph k-Colorability problem can be reduced to a special case of WSP in which the workflow specification only includes separation-of-duty constraints [14]. Clearly, it is important to be able to determine whether a workflow specification is satisfiable at design time. Equally, when users select steps to execute in a workflow instance, it is essential that the access control mechanism can determine whether (a) the user is authorized, (b) allowing the user to execute the step would render the instance unsatisfiable. Thus, the access control mechanism must incorporate an algorithm to solve WSP, and that algorithm needs to be as efficient as possible.

Wang and Li [14] observed that, in practice, the number k of steps in a workflow will be small, relative to the size of the input to WSP; specifically, the number of users is likely to be an order of magnitude greater than the number of steps. This observation led them to set k as the parameter and to study the problem using tools from parameterized complexity. In doing so, they proved that the problem is *fixed-parameter tractable* (FPT) for simple classes of constraints. However, Wang and Li also showed that for many types of constraints the problem is fixed-parameter *intractable* (unless FPT $\neq$ W[1] is false). Hence, it is important to be able to identify those types of practical constraints for which WSP is FPT.

Recent research has made significant progress in understanding the fixed-parameter tractability of WSP. In particular, Cohen *et al.* [5] introduced the notion of patterns and, using it, proved that WSP is FPT (irrespective of the authorization policy) if all constraints in the specification are UI. This result is significant because most constraints in the literature – including separation-of-duty, cardinality and counting constraints – are UI [5]. Using a modified pattern approach, Karapetyan *et al.* [11] provided both a short proof that WSP with only UI constraints is FPT and a very efficient algorithm for WSP with UI constraints.

However, it is known that not all constraints that may be useful in practice are UI. Consider a situation where the set of users is partitioned into groups (such as departments or teams) and we wish to define constraints on the groups, rather than users. In our purchase order example, suppose each user belongs to a specific department. Then it would be reasonable to require that steps s_1 and s_2 are performed by different users belonging to the same department. There is little work in the literature on constraints of this form, although prior work has recognized that such constraints are likely to be important in practice [7, 14], and it has been shown that such constraints present additional difficulties when incorporated into WSP [9].

In this paper, we extend the notion of a UI constraint to that of a *class-independent* (CI) constraint. In particular, every UI constraint is an instance of a CI constraint. Our second contribution is to demonstrate that patterns for UI constraints [5] can be generalized to patterns for CI constraints. The resulting algorithm, using these new patterns, remains FPT (irrespective of the authorization policy), although its running time is slower than that of the algorithm for WSP with UI constraints only. In short, our first two contributions identify a large class of constraints for which WSP is shown to be FPT, and subsume prior work in this area [9, 5, 14]. Our final contribution is an implementation of our algorithm in order to

investigate whether the theoretical advantages implied by its fixed-parameter tractability can be realized in practice. We compare our FPT algorithm with SAT4J, an off-the-shelf pseudo-Boolean (PB) SAT solver. The results of our experiments suggest that our FPT algorithm enjoys some significant advantages over SAT4J for hard instances of WSP.

In the next section, we define WSP and UI constraints in more formal terms, discuss related work in more detail, and introduce the notion of class-independent constraints. In Sections 3 and 4, we state and prove a number of technical results that underpin the algorithm for solving WSP with class-independent constraints. We describe the algorithm and establish its worst-case complexity in Section 5. In Section 6, we describe our experimental methods and report the results of our experiments. We conclude in Section 7.

Proofs of results marked with a $\star$ are given in Appendix A of the full version [8] of the paper. We also provide some details about the implementation of our algorithm in Appendix B of [8]. In the main body of the paper, we focus on the case of a single non-trivial partition of the user set. In Appendix C of [8], we generalize our approach to handle multiple (nested) partitions of the user set. (Such partitions can be used to model hierarchical organizational structures, which can be useful in practice [9].)

2 Workflow Satisfiability

Let $S = \{s_1, \ldots, s_k\}$ be a set of *steps*, let $U = \{u_1, \ldots, u_n\}$ be a set of *users* in a workflow specification, and let $k \leq n$. We are interested in assigning users to steps subject to certain constraints. In other words, among the set $\Pi(S, U)$ of functions from S to U, there are some that represent "legitimate" assignments of steps to users and some that do not.

The legitimacy or otherwise of an assignment is determined by the authorization policy and the constraints that complete the workflow specification. Let $\mathcal{A} = \{A(u) : u \in U\}$ be a set of *authorization lists*, where $A(u) \subseteq S$ for each $u \in U$, and let C be a set of *(workflow) constraints*. A *constraint* $c \in C$ may be viewed as a pair (T, Θ), where $T \subseteq S$ is the *scope* of c and Θ is a set of functions from T to U, specifying the assignments of steps in T to users in U that satisfy the constraint. In practice, we do not enumerate all the elements of Θ. Instead, we define its members implicitly using some constraint-specific syntax. In particular, we write (s, s', ρ), where $s, s' \in S$ and ρ is a binary relation defined on U, to denote a constraint that has scope $\{s, s'\}$ and is satisfied by any plan $\pi : S \to U$ such that $(\pi(s), \pi(s')) \in \rho$. Thus $(s, s', \neq)$, for example, requires s and s' to be performed by different users (and so represents a separation-of-duty constraint). Also $(s, s', =)$ states that s and s' must be performed by the same user (a binding-of-duty constraint).

2.1 The Workflow Satisfiability Problem

A *plan* is a function in $\Pi(S, U)$. Given a *workflow* $W = (S, U, \mathcal{A}, C)$, a plan π is *authorized* if for all $s \in S$, $s \in A(\pi(s))$, i.e. the user assigned to s is authorized for s. A plan π is *eligible* if for all $(T, \Theta) \in C$, $\pi|_T \in \Theta$, i.e. every constraint is satisfied. A plan π is *valid* if it is both authorized and eligible. In the *workflow satisfiability problem (WSP)*, we are given a workflow (specification) W, and our aim is to decide whether W has a valid plan. If W has a valid plan, W is *satisfiable*; otherwise, W is *unsatisfiable*.

Note that WSP is, in fact, the conservative CSP (i.e., CSP with unary constraints corresponding to step authorizations in the WSP terminology). However, unlike a typical instance of CSP, where the number of variables is significantly larger than the number of values, a typical instance of WSP has many more values (i.e., users) than variables (i.e., steps).

We assume that in all instances of WSP we consider, all constraints can be checked in time polynomial in n. Thus it takes polynomial time to check whether any plan is eligible. The correctness of our algorithm is unaffected by this assumption, but using constraints not checkable in polynomial time would naturally affect the running time.

▶ **Example 1.** Consider the following instance W' of WSP. The step and user sets are $S = \{s_1, s_2, s_3, s_4\}$ and $U = \{u_1, u_2, u_3, u_4, u_5\}$. The authorization lists are $A(u_1) = \{s_1, s_2, s_3, s_4\}$, $A(u_2) = \{s_1\}$, $A(u_3) = \{s_2\}$, $A(u_4) = A(u_5) = \{s_3, s_4\}$. The constraints are $(s_1, s_2, =)$, $(s_2, s_3, \neq)$, $(s_3, s_4, \neq)$, and $(s_4, s_1, \neq)$. Observe that $\pi' : S \to U$ with $\pi'(s_1) = \pi'(s_2) = u_1$, $\pi'(s_3) = u_5$ and $\pi'(s_4) = u_4$ satisfies all constraints and authorizations, and thus π' is a valid plan for W'. Therefore, W' is satisfiable.

2.2 Constraints using Equivalence Relations

Crampton *et al.* [9] introduced constraints defined in terms of an equivalence relation $\sim$ on U: a plan π satisfies constraint $(s, s', \sim)$ if $\pi(s) \sim \pi(s')$ (and satisfies constraint $(s, s', \nsim)$ if $\pi(s) \nsim \pi(s')$). Hence, we could, for example, specify the pair of constraints $(s, s', \neq)$ and $(s, s', \sim)$, which, collectively, require that s and s' are performed by different users that belong to the same equivalence class. As we noted in the introduction, such constraints are very natural in the context of organizations that partition the set of users into departments, groups or teams.

Moreover, Crampton *et al.* [9] demonstrated that "nested" equivalence relations can be used to model hierarchical structures within an organization[1] and to define constraints on workflow execution with respect to those structures. More formally, an equivalence relation $\sim$ is said to be a *refinement* of an equivalence relation $\approx$ if $x \sim y$ implies $x \approx y$. In particular, given an equivalence relation $\sim$, $=$ is a refinement of $\sim$. Crampton *et al.* proved that WSP remains FPT when some simple extensions of constraints $(s, s', \sim)$ and $(s, s', \nsim)$ are included [9, Theorem 5.4]. Our extension of constraints $(s, s', \sim)$ and $(s, s', \nsim)$ is much more general: it is similar to generalizing simple constraints $(s, s', =)$ and $(s, s', \neq)$ to the wide class of UI constraints. This leads, in particular, to a significant generalization of Theorem 5.4 in [9].

Let $c = (T, \Theta)$ be a constraint and let $\sim$ be an equivalence relation on U. Let $U^{\sim}$ denote the set of equivalence classes induced by $\sim$ and let $u^{\sim} \in U^{\sim}$ denote the equivalence class containing u. Then, for any function $\pi : S \to U$, we may define the function $\pi^{\sim} : S \to U^{\sim}$, where $\pi^{\sim}(s) = (\pi(s))^{\sim}$. In particular, $\sim$ induces a set of functions $\Theta^{\sim} = \{\theta^{\sim} : \theta \in \Theta\}$.

▶ **Example 2.** Continuing from Example 1, suppose $U^{\sim}$ consists of two equivalence classes $U_1 = \{u_1, u_2, u_5\}$ and $U_2 = \{u_3, u_4\}$. Let us add to W' another constraint $(s_1, s_4, \sim)$ (s_1 and s_4 must be assigned users from the same equivalence class) to form a new instance W'' of WSP. Then plan π' does not satisfy the added constraint and so π' is not valid for W''. However, $\pi'' : S \to U$ with $\pi''(s_1) = \pi''(s_2) = u_1$, $\pi''(s_3) = u_4$ and $\pi''(s_4) = u_5$ satisfies all constraints and authorizations, and thus π'' is valid for W''. Here $(\pi'')^{\sim}(s_1) = (\pi'')^{\sim}(s_2) = (\pi'')^{\sim}(s_4) = U_1$ and $(\pi'')^{\sim}(s_3) = U_2$.

Given an equivalence relation $\sim$ on U, we say that a constraint $c = (T, \Theta)$ is *class-independent (CI) for* $\sim$ if $\theta^{\sim} \in \Theta^{\sim}$ implies $\theta \in \Theta$, and for any permutation $\phi : U^{\sim} \to U^{\sim}$, $\theta^{\sim} \in \Theta^{\sim}$ implies $\phi \circ \theta^{\sim} \in \Theta^{\sim}$. In other words, if a plan $\pi : s \mapsto \pi(s)$ satisfies a constraint c,

[1] Many organizations exhibit nested hierarchical structure. For example, the academic parts of many universities are divided into faculties/schools which are divided into departments.

which is class-independent for $\sim$, then for each permutation ϕ of classes in $U^{\sim}$ if we replace $\pi(s)$ by any user in the class $\phi(\pi(s)^{\sim})$ for every step s, then the new plan will satisfy c.

We say a constraint is *user-independent (UI)* if it is CI for $=$. In other words, if a plan $\pi : s \mapsto \pi(s)$ satisfies a UI constraint c and we replace any user in $\{\pi(s) : s \in S\}$ by an arbitrary user such that the replacement users are all distinct, then the new plan satisfies c.

We conclude this section with a claim whose simple proof is omitted.

▶ **Proposition 3.** *Given two equivalence relations $\sim$ and $\approx$ such that $\sim$ is a refinement of $\approx$, and any plan $\pi : S \to U$, $\pi^{\sim}(s) = \pi^{\sim}(s')$ implies $\pi^{\approx}(s) = \pi^{\approx}(s')$.*

3 Plans and Patterns

In what follows, unless specified otherwise, we will consider the equivalence relation $=$ along with another fixed equivalence relation $\sim$. We will write $[m]$ to denote the set $\{1, \ldots, m\}$ for any integer $m \geqslant 1$. We assume that all constraints are either UI or CI for $\sim$. For brevity, we will refer to constraints that are CI for $\sim$ as simply *CI*. We consider only two equivalence relations for simplicity of presentation, but our results below can be generalized to any sequence $\sim_1, \ldots, \sim_l$ of equivalence relations such that $\sim_{i+1}$ is a refinement of $\sim_i$ for all $i \in [l-1]$, see Appendix C in the full version [8] of the paper. It is important to keep in mind that we put *no restrictions on authorizations*.

We will represent groups of plans as *patterns*. The intuition is that a pattern defines a partition of the set of steps relevant to a set of constraints. For instance, suppose that we only have UI constraints. Then a pattern specifies which sets of steps are to be assigned to the same user. A pattern assigns an integer to each step and those steps that are labelled by the same integer will be mapped to the same user. A pattern p defines an equivalence relation $\sim_p$ on the set of steps (where $s \sim_p s'$ if and only if s and s' are assigned the same label). Moreover, this pattern can be used to define a plan by mapping each of the equivalence classes induced by $\sim_p$ to a different user. Since we only consider UI constraints, the identities of the users are irrelevant (provided they are distinct). Conversely, any plan $\pi : S \to U$ defines a pattern: s and s' are labelled with the same integer if and only if $\pi(s) = \pi(s')$. And if π satisfies a UI constraint c, then any other plan with the same pattern will also satisfy c. We can extend this notion of a pattern to CI constraints where entries in the pattern encode equivalence classes of users instead of single users.

More formally, let $W = (S, U, \mathcal{A}, C = C_{=} \cup C_{\sim})$ be a workflow, where $C_{=}$ is a set of UI constraints and $C_{\sim}$ is a set of CI constraints. Let $p_{=} = (x_1, \ldots, x_k)$ where $x_i \in [k]$ for all $i \in [k]$. We say that $p_{=}$ is a *UI-pattern* for a plan π if $x_i = x_j \Leftrightarrow \pi(s_i) = \pi(s_j)$, for all $i, j \in [k]$, and $p_{=}$ is *eligible for* $C_{=}$ if any plan π with $p_{=}$ as its UI-pattern is eligible for $C_{=}$.

In Example 2, $C_{=} = \{(s_1, s_2, =), (s_2, s_3, \neq), (s_3, s_4, \neq), (s_1, s_4, \neq)\}$ and $C_{\sim} = \{(s_1, s_4, \sim)\}$. Tuples $(1, 1, 2, 3)$ and $(2, 2, 4, 3)$ are UI-patterns for plan π'' of Example 2.

▶ **Proposition 4** (⋆). *Let $p_{=}$ be a UI-pattern for a plan π. Then $p_{=}$ is eligible for $C_{=}$ if and only if π is eligible for $C_{=}$.*

Let $p_{\sim} = (y_1, \ldots, y_k)$, where $y_i \in [k]$ for all $i \in [k]$. We say that $p_{\sim}$ is a *CI-pattern* for a plan π if $y_i = y_j \Leftrightarrow \pi^{\sim}(s_i) = \pi^{\sim}(s_j)$, for all $i, j \in [k]$, and $p_{\sim}$ is *eligible for* $C_{\sim}$ if any plan π with $p_{\sim}$ as its CI-pattern is eligible for $C_{\sim}$. For example, $(1, 1, 2, 1)$ and $(2, 2, 4, 2)$ are CI-patterns for plan π'' of Example 2. The next result is a generalization of Proposition 4.

▶ **Proposition 5** (⋆). *Let $p_{\sim}$ be a CI-pattern for a plan π. Then $p_{\sim}$ is eligible for $C_{\sim}$ if and only if π is eligible for $C_{\sim}$.*

Now let $p = (p_=, p_\sim)$ be a pair containing a UI-pattern and an CI-pattern. Then we call p a *(UI, CI)-pattern*. We say that p is a (UI, CI)-pattern for π if $p_=$ is a UI-pattern for π and $p_\sim$ is a CI-pattern for π. We say that p is *eligible for* $C = C_= \cup C_\sim$ if $p_=$ is eligible for $C_=$ and $p_\sim$ is eligible for $C_\sim$. The following two results follow immediately from Propositions 4 and 5 and definitions of UI- and CI-patterns.

▶ **Lemma 6.** *Let $p = (p_=, p_\sim)$ be a (UI, CI)-pattern for a plan π. Then p is eligible for $C = C_= \cup C_\sim$ if and only if π is eligible for C.*

▶ **Proposition 7.** *There is a (UI, CI)-pattern p for every plan π.*

We say a (UI, CI)-pattern p is *realizable* if there exists a plan π such that π is authorized and p is a (UI, CI)-pattern for π. Given the above results, in order to solve a WSP instance with user- and class-independent constraints, it is enough to decide whether there exists a (UI, CI)-pattern p such that (i) p is realizable, and (ii) p is eligible (and hence π is eligible) for $C = C_= \cup C_\sim$.

We will enumerate all possible (UI, CI)-patterns, and for each one check whether the two conditions hold. We defer the explanation of how to determine whether p is realizable until Sec. 4. We now show it is possible to check whether a (UI, CI)-pattern $p = (p_=, p_\sim)$ is eligible in time polynomial in the input size N. Indeed, in polynomial time, we can construct plans $\pi_=$ and $\pi_\sim$ with patterns $p_=$ and $p_\sim$, respectively, where $\pi_=(s_i) = \pi_=(s_j)$ if and only if $x_i = x_j$ and $\pi_\sim(s_i) \sim \pi_\sim(s_j)$ if and only if $y_i = y_j$. (In particular, we can select a representative user from each equivalence class in $U^\sim$.) By Lemma 6 and Propositions 4 and 5, p is eligible if and only if both $\pi_=$ and $\pi_\sim$ are eligible. By our assumption before Example 1, eligibility of both $\pi_=$ and $\pi_\sim$ can be checked in polynomial time.[2] Note, however, that $\pi_=$ and $\pi_\sim$ may be different plans, so this simple check for eligibility does not give us a check for realizability of p.

4 Checking Realizability

A *partial plan* π is a function from a subset T of S to U. In particular, a plan is a partial plan. To avoid confusion with partial plans, sometimes we will call plans *complete plans*. We can easily extend the definitions of *eligible, authorized* and *valid* plans to partial plans: the only difference is that we only consider authorizations for steps in T and constraints with scope being a subset of T.

We also define *partial patterns*. For a UI or CI-pattern $q = (x_1, \ldots, x_k)$ and a subset $T \subseteq S$, let the pattern $q|_T = (z_1, \ldots, z_k)$, where $z_i = x_i$ if $s_i \in T$, and $z_i = 0$ otherwise. We say that $p|_T$ is a (UI, CI)-pattern for a partial plan $\pi : T \to U$ if $p|_T$ with all coordinates with 0 values removed is a (UI, CI)-pattern for π. We therefore have that if p is a (UI, CI)-pattern for a plan π, then $p|_T$ is a (UI, CI)-pattern for π restricted to T.

Let $p = (p_= = (x_1, \ldots, x_k), p_\sim = (y_1, \ldots, y_k))$ be a (UI, CI)-pattern. We say that p is *consistent* if $x_i = x_j \Rightarrow y_i = y_j$ for all $i, j \in [k]$. Recall that if p is the (UI, CI)-pattern for π, then $x_i = x_j \Leftrightarrow \pi(s_i) = \pi(s_j)$, and $y_i = y_j \Leftrightarrow \pi^\sim(s_i) = \pi^\sim(s_j)$. Thus Proposition 3 implies that if p is the (UI, CI)-pattern for any plan then p is consistent. Henceforth, we will only consider (UI, CI)-patterns that are consistent.

Given a (UI, CI)-pattern $(p_=, p_\sim)$, we must determine whether this (UI, CI)-pattern can be realized, given the authorization lists defined on users. The patterns $p_=$ and $p_\sim$ define

[2] Clearly, it is not hard to check eligibility of p without explicitly constructing $\pi_=$ and $\pi_\sim$, as is done in our algorithm implementation, described in Appendix B of [8].

two sets of equivalence classes on S: s_i and s_j are in the same equivalence class of S defined by $p_=$ ($p_\sim$, respectively) if and only if $x_i = x_j$ ($y_i = y_j$, respectively).

Moreover each equivalence class induced by $p_\sim$ is partitioned by equivalence classes induced by $p_=$. We must determine whether there exists a plan $\pi : S \to U$ that simultaneously (i) has UI-pattern $p_=$; (ii) has CI-pattern $p_\sim$; and (iii) assigns an authorized user to each step. Informally, our algorithm for checking realizability computes two things.

- For each pair (T, V), where $T \subseteq S$ is an equivalence class induced by $p_\sim$ and $V \subseteq U$ is an equivalence class induced by $\sim$, whether there exists an injective mapping from the equivalence classes in T induced by $p_=$ to authorized users in V. We call such a mapping a *second-level* mapping.
- Whether there exists an injective mapping f from the set of equivalence classes induced by $p_\sim$ to the set of equivalence classes induced by $\sim$ such that $f(T) = V$ if and only if there exists a second-level mapping from T to V. We call f a *top-level* mapping.

If a top-level mapping exists, then, by construction, it can be "deconstructed" into authorized partial plans defined by second-level mappings. We compute top- and second-level mappings using matchings in bipartite graphs, as described below.

The Top-level Bipartite Graph. The UI-pattern $p_= = (x_1, \ldots, x_k)$ induces an equivalence relation on $S = \{s_1, \ldots, s_k\}$, where s_i and s_j are equivalent if and only if $x_i = x_k$. Let $\mathcal{S} = \{S_1, \ldots, S_l\}$ be the set of equivalence classes of S under this relation. Similarly, the CI-pattern $p_\sim = (y_1, \ldots, y_k)$ induces an equivalence relation on S, where s_i, s_j are equivalent if and only if $y_i = y_j$. Let $\mathcal{T} = \{T_1, \ldots, T_m\}$ be the equivalence classes under this relation. Observe that since p is consistent, we have $k \geq l \geq m$ and for any S_i, T_j, either $S_i \subseteq T_j$ or $S_i \cap T_j = \emptyset$.

▶ **Definition 8.** Given a (UI, CI)-pattern $p = (p_=, p_\sim)$, the *top-level bipartite graph* G_p is defined as follows. Let the partite sets of G_p be $\mathcal{T}$ and $U^\sim$. For each $T_r \in \mathcal{T}$ and class $u^\sim$, we have an edge between T_r and $u^\sim$ if and only if there exists an authorized partial plan $\pi_r : T_r \to u^\sim$ such that $p_=|_{T_r}$ is a UI-pattern for π_r.

▶ **Lemma 9.** *If a (UI, CI)-pattern $p = (p_=, p_\sim)$ is realizable, then G_p has a matching covering $\mathcal{T}$.*

Proof. Let π be an authorized plan such that p is a (UI, CI)-pattern for π. As $p_\sim$ is a CI-pattern for π, we have that for each $T_r \in \mathcal{T}$ and all $s_i, s_j \in T_r$, $\pi^\sim(s_i) = \pi^\sim(s_j)$. Therefore $\pi(T_r) \subseteq u^\sim$ for some $u \in U$. Let $u_r^\sim$ be this equivalence class for each T_r. As $p_\sim$ is a CI-pattern for π, we have that for all $r \neq r'$ and any $s_i \in T_r, s_j \in T_{r'}$, $\pi^\sim(s_i) \neq \pi^\sim(s_j)$. It follows that $u_r^\sim \neq u_{r'}^\sim$ for any $r \neq r'$.

Let $M = \{T_r u_r^\sim \in E(G_p) : T_r \in \mathcal{T}\}$. As $u_r^\sim \neq u_{r'}^\sim$ for any $r \neq r'$ we have that M is a matching that covers $\mathcal{T}$. It remains to show that M is a matching of G_p covering $\mathcal{T}$, i.e. that $T_r u_r^\sim$ is an edge in G_p for each T_r. For each $T_r \in \mathcal{T}$, let π_r be π restricted to T_r. Then π_r is a function from T_r to $u_r^\sim$. As π is authorized, π_r is also authorized. As $p_=$ is a UI-pattern for π, we have that $p_=|_{T_r}$ is a UI-pattern for π_r. Therefore π_r satisfies all the conditions for there to be an edge $T_r u_r^\sim$ in G_p. ◀

We have shown that for any (UI, CI)-pattern to be realizable, it must be consistent and its top-level bipartite graph must have a matching covering $\mathcal{T}$. We will now show that these necessary conditions are also sufficient.

▶ **Lemma 10 (⋆).** *Let $p = (p_= = (x_1, \ldots, x_k), p_\sim = (y_1, \ldots, y_k))$ be a (UI, CI)-pattern which is consistent, and such that G_p has a matching covering $\mathcal{T}$. Then p is realizable.*

The Second-level Bipartite Graph. For each (UI, CI)-pattern $p = (p_=, p_\sim)$, we need to construct the graph G_p and decide whether it has a matching covering $\mathcal{T}$, in order to decide whether p is realizable. Given G_p, a maximum matching can be found in polynomial time using standard techniques, but constructing G_p itself is non-trivial. For each potential edge $T_r u^\sim$ in G_p, we need to decide whether there exists an authorized partial plan $\pi_r : T_r \to u^\sim$ such that $p_=|_{T_r}$ is a UI-pattern for π_r. We can decide this by constructing another bipartite graph, $G_{T_r u^\sim}$. Recall that $\mathcal{S} = \{S_1, \ldots, S_l\}$ is a partition of S into equivalence classes, where s_i, s_j are equivalent if $x_i = x_j$, and for each $S_h \in \mathcal{S}$, either $S_h \subseteq T_r$ or $S_h \cap T_r = \emptyset$. Define $\mathcal{S}_r = \{S_h : S_h \subseteq T_r\}$.

▶ **Definition 11.** Given a (UI, CI)-pattern $p = (p_= = (x_1, \ldots, x_k), p_\sim = (y_1, \ldots, y_k))$, a set $T_r \in \mathcal{T}$ and equivalence class $u^\sim \in U^\sim$, the *second-level bipartite graph* $G_{T_r u^\sim}$ is defined as follows: Let the partite sets of G be $\mathcal{S}_r$ and $u^\sim$ and for each $S_h \in \mathcal{S}_r$ and $v \in u^\sim$, we have an edge between S_h and v if and only if v is authorized for all steps in S_h.

▶ **Lemma 12 (⋆).** *Given $T_r \in \mathcal{T}$, $u^\sim \in U^\sim$, the following conditions are equivalent.*

- *There exists an authorized partial plan $\pi : T_r \to u^\sim$ such that $p_=|_{T_r}$ is a UI-pattern for π.*
- *$G_{T_r u^\sim}$ has a matching that covers $\mathcal{S}_r$.*

5 FPT Algorithm

Our FPT algorithm generates (UI, CI)-patterns p in a backtracking manner as follows. (Its implementation is described in Appendix B of [8].) It first generates partial patterns $p_= = (x_1, \ldots, x_k)$, where the coordinates $x_i = 0$ are assigned one by one to integers in $[k']$, where $k' = \max_{1 \le j \le k}\{x_j\} + 1$. The algorithm checks that the pattern $p_=$ does not violate any constraints whose scope contain the corresponding step s_i. If an eligible pattern $p_= = (x_1, \ldots, x_k)$ has been completed (i.e., $x_j \neq 0$ for each $j \in [k]$), the partial patterns $p_\sim = (y_1, \ldots, y_k)$ are generated as above but with a difference: the algorithm ensures the consistency condition. If an eligible (UI, CI)-pattern p has been constructed, a procedure constructing bipartite graphs and searching for matchings in them as described in Section 4 decides whether p is realizable.

▶ **Theorem 13.** *We can solve WSP with UI and CI constraints in $O^*(4^{k \log_2 k})$ time.*

Proof Sketch. Our algorithm is correct by Proposition 7 and the fact that every complete (UI, CI)-pattern can be generated. It remains to estimate the running time.

If $p_\sim$ in our algorithm were generated as $p_=$, i.e., consistency were not taken into consideration, the search tree $\mathcal{T}$ of our algorithm (nodes are partial (UI, CI)-patterns) would have at least as many nodes as the actual search tree of our algorithm. Observe that each internal node (corresponding to an incomplete (UI, CI)-pattern) in $\mathcal{T}$ has at least two children, and each leaf in this tree corresponds to a complete (UI, CI)-pattern. Thus, the total number of partial (UI, CI)-patterns considered by our algorithm is less than twice the number of complete (UI, CI)-patterns, which is $k^{2k} = 4^{k \log k}$ as each of $2k$ coordinates takes values in $[k]$.

We have to compute a matching in the top-level bipartite graph and matchings for each second-level bipartite graph. The number of second-level bipartite graphs is bounded above by nk (since the number of equivalence classes in U and S cannot exceed n and k, respectively). We can compute a maximum matching in time polynomial in the number of vertices in the top- and second-level bipartite graphs (which is bounded in all our graphs by $n + k$). The result follows. ◀

6 Algorithm Implementation and Computational Experiments

There can be a huge difference between an algorithm in principle and its actual implementation as a computer code, e.g., see [13]. We have implemented the new pattern-backtracking FPT algorithm and a reduction to the pseudo-Boolean satisfiability (PB SAT) problem in C++, using SAT4J [12] as a pseudo-Boolean SAT solver. Reductions from WSP constraints to PB ones were done similarly to those in [4, 6, 11]. Our FPT algorithm extends the pattern-backtracking framework of [11] in a nontrivial way, for details see Appendix B of [8].

In this section we describe a series of experiments that we ran to test the performance of our FPT algorithm against that of SAT4J. Due to the difficulty of acquiring real-world workflow instances, we generate and use synthetic data to test our new FPT algorithm and reduction to the PB SAT problem (as in similar experimental studies [6, 11, 14]). All our experiments use a MacBook Pro computer having a 2.6 GHz Intel Core i5 processor, 8 GB 1600 MHz DDR3 RAM and running Mac OS X 10.9.5.

We generate a number of random WSP instances using not-equals (i.e, constraints of the form $(s, s', \neq)$), equivalence and non-equivalence constraints (i.e., constraints of the types $(s, s', \sim)$ and $(s, s', \nsim)$), and at-most constraints. An *at-most constraint* is a UI constraint that restricts the number of users that may be involved in the execution of a set of steps. It is, therefore, a form of cardinality constraint and imposes a loose form of "need-to-know" constraint on the execution of a workflow instance, which can be important in certain business processes. An at-most constraint may be represented as a tuple $(t, Q, \leqslant)$, where $Q \subseteq S$, $1 \leqslant t \leqslant |Q|$, and is satisfied by any plan that allocates no more than t users in total to the steps in Q. In all our at-most constraints $t = 3$ and $|Q| = 5$ as in [6, 11].

6.1 Experimental Parameters and Instance Generation

We summarize the parameters we use for our experiments in Table 1. Values of k, n and r were chosen that seemed appropriate for real-world workflow specifications. The values of the other parameters were determined by preliminary experiments designed to identify "challenging" instances of WSP: that is, instances that were neither very lightly constrained nor very tightly constrained. Informally, it is relatively easy to determine that lightly constrained instances are satisfiable and that tighly constrained instances are unsatisfiable. Thus the instances we use in our experiments are (very approximately) equally likely to be satisfiable or unsatisfiable. In particular, by varying the numbers of at-most constraints and constraints of the form $(s, s', \nsim)$, we are able to generate a set of instances with the desired characteristics (as shown by the results in Table 2).

A constraint $(s, s', \nsim)$ implies the existence of a constraint $(s, s', \neq)$, so we do not vary the number of not-equals a great deal (in contrast to existing work in the literature [6]). Informally, a constraint $(s, s', \sim)$ reduces the difficulty of finding a valid plan. Thus, given our desire to investigate challenging instances, we do not use very many of these constraints.

All the constraints, authorizations, and equivalence classes of users are generated for each instance separately, uniformly at random. The random generation of authorizations, not-equals, and at-most constraints uses existing techniques [6]. The generation of equivalence and non-equivalence constraints has to be controlled to ensure that an instance is not trivially unsatisfiable. In particular, we must discard a constraint of the form $(s, s', \nsim)$ if we have already generated a constraint of the form $(s, s', \sim)$. The equivalence classes of the user set are generated by enumerating the user set and then splitting the list into contiguous sublists. The number of elements in each sublist varies between 3 and 7 (chosen uniformly at random and adjusted, where necessary, so that the total number of members in the r sub-lists is n).

Table 1 Parameters used in our experiments.

Parameter		Values
Number of steps k		$20, 25, 30$
Number of users n		$10k$
Number of user equivalence classes r		$2k$
Number of constraints $(s, s', \neq)$	$k = 20$	$20, 25$
	$k = 25$	$25, 30$
	$k = 30$	$30, 35$
Number of constraints $(s, s', \sim)$	$k = 20$	0
	$k = 25$	1
	$k = 30$	2
Number of constraints $(s, s', \nsim)$	$k = 20$	$10, 15, 20, 25, 30$
	$k = 25$	$15, 20, 25, 30, 35$
	$k = 30$	$20, 25, 30, 35, 40$
Number of at-most constraints	$k = 20$	$10, 15, 20, 25, 30, 35, 40$
	$k = 25$	$15, 20, 25, 30, 35, 40, 45$
	$k = 30$	$20, 25, 30, 35, 40, 45, 50$

6.2 Results and Evaluation

We adopt the following labelling convention for our test instances: $a.b.c.d$ denotes an instance with a not-equals constraints, b at-most constraints, c equivalence constraints, and d non-equivalence constraints (as used in the first and fourth columns of Table 2, for instances with $k = 25$ and $k = 30$, respectively). In our experiments we compare the run-times and outcomes of SAT4J (having reduced the WSP instance to a PB SAT problem instance) and our FPT algorithm, which we will call PBA4CI (pattern-based algorithm for class-independent constraints). Table 2 shows some detailed results of our experiments (the results for $k = 20$ were excluded due to the space limit). We record whether an instance is solved, indicating a satisfiable instance with a 'Y' and an unsatisfiable instance with a 'N'; instances that were not solved are indicated by a question mark. PBA4CI reaches a conclusive decision (Y or N) for every test instance, whereas SAT4J fails to reach such a decision for some instances, typically because the machine runs out of memory. The table also records the time (in seconds) taken for the algorithms to run on each instance. We would expect that the time taken to solve an instance would depend on whether the instance is satisfiable or not, and this is confirmed by the results in the table.

In total, the experiments cover 210 randomly generated instances, 70 instances for each number of steps, $k \in \{20, 25, 30\}$. PBA4CI successfully solves all of the instances, while SAT4J fails on almost 40% of the instances (mostly unsatisfiable ones). In terms of CPU time, SAT4J is more efficient only on 5 instances (2.4%) in total: 1 for 20 steps, 1 for 25 steps, and 3 for 30 steps, all of which are lightly constrained. For these instances PBA4CI has to generate a large number of patterns in the search space before it finds a solution.

Overall, PBA4CI is clearly more effective and efficient than SAT4J on these instances. Table 3 put in Appendix B of [8] shows the summary statistics for all the experiments. The numbers of unsolved instances by SAT4J are indicated in parenthesis. For average CPU time values, we assume that the running time on the unsolved instances can be considered as a lower bound on the time required to solve them. Therefore average time values in Table 3 take into consideration unsolved instances for SAT4J: they are estimated lower bounds on its

Table 2 Results for $k = 25$ and 30. Time in seconds. Y,N,? mean satisfied, unsatisfied, unsolved.

Instance	SAT4J		PBA4CI		Instance	SAT4J		PBA4CI	
	$k = 25$					$k = 30$			
25.15.1.15	Y	2.62	Y	2.464	30.20.2.20	Y	2.72	Y	50.804
25.20.1.15	Y	22.38	Y	0.010	30.25.2.20	Y	271.78	Y	2.323
25.25.1.15	Y	11.03	Y	0.010	30.30.2.20	?	2,141.60	Y	2.946
25.30.1.15	Y	35.54	Y	0.040	30.35.2.20	?	2,250.02	N	0.412
25.35.1.15	N	1,439.94	N	0.075	30.40.2.20	?	1,942.57	N	2.238
25.40.1.15	?	2,088.06	N	0.033	30.45.2.20	?	2,198.02	N	2.171
25.45.1.15	Y	113.37	Y	0.022	30.50.2.20	?	2,580.81	N	0.494
25.15.1.20	Y	1.52	Y	111.799	30.20.2.25	Y	4.18	Y	237.604
25.20.1.20	Y	7.77	Y	0.024	30.25.2.25	Y	76.41	Y	0.789
25.25.1.20	Y	297.39	Y	0.065	30.30.2.25	?	2,288.07	N	0.401
25.30.1.20	?	2,273.56	N	0.033	30.35.2.25	Y	1,364.66	Y	0.238
25.35.1.20	Y	48.29	Y	0.067	30.40.2.25	?	2,383.92	N	0.775
25.40.1.20	N	105.48	N	0.045	30.45.2.25	?	1,743.87	N	0.394
25.45.1.20	?	2,105.61	N	0.031	30.50.2.25	?	2,385.39	N	0.218
25.15.1.25	Y	14.40	Y	0.014	30.20.2.30	Y	35.40	Y	0.071
25.20.1.25	Y	80.25	Y	0.021	30.25.2.30	Y	9.37	Y	1.063
25.25.1.25	?	2,284.78	N	0.023	30.30.2.30	N	1,632.51	N	0.347
25.30.1.25	N	442.91	N	0.237	30.35.2.30	Y	803.50	Y	0.029
25.35.1.25	?	2,188.01	N	0.060	30.40.2.30	?	2,022.71	N	0.981
25.40.1.25	?	2,293.77	N	0.043	30.45.2.30	?	1,902.84	N	1.501
25.45.1.25	?	2,041.02	N	0.144	30.50.2.30	?	1,730.93	N	0.467
25.15.1.30	Y	3.22	Y	0.011	30.20.2.35	Y	24.12	Y	0.453
25.20.1.30	Y	240.59	Y	0.014	30.25.2.35	Y	456.51	Y	0.085
25.25.1.30	Y	66.74	Y	0.050	30.30.2.35	N	1,817.76	N	1.088
25.30.1.30	?	2,301.75	N	0.088	30.35.2.35	?	1,949.77	N	0.111
25.35.1.30	N	1,562.30	N	0.023	30.40.2.35	?	2,115.32	N	0.551
25.40.1.30	?	2,332.07	N	0.127	30.45.2.35	?	1,535.57	N	0.118
25.45.1.30	N	950.25	N	0.040	30.50.2.35	?	1,647.41	N	0.454
25.15.1.35	Y	10.57	Y	0.014	30.20.2.40	?	3,088.54	N	0.729
25.20.1.35	N	218.70	N	0.166	30.25.2.40	?	1,746.81	Y	0.542
25.25.1.35	Y	37.87	Y	0.012	30.30.2.40	?	2,350.01	Y	0.949
25.30.1.35	?	2,421.30	N	0.054	30.35.2.40	?	1,857.27	N	0.576
25.35.1.35	N	1,524.68	N	0.022	30.40.2.40	?	1,938.63	N	0.221
25.40.1.35	N	1,001.67	N	0.028	30.45.2.40	?	2,159.50	N	0.209
25.45.1.35	?	1,974.05	N	0.034	30.50.2.40	?	1,815.15	N	0.337

average time performance. As the number of steps k increases, SAT4J fails more frequently and is unable to reach a conclusive decision for more than 65% of instances when $k = 30$, some of which are satisfiable. However, SAT4J is clearly more efficient (and effective) on satisfiable instances than on the unsatisfiable ones, while for PBA4CI the converse is true. This can be explained by very different search strategies used by the solvers.

7 Conclusion

We have introduced the concept of a class-independent constraint, which significantly generalizes user-independent constraints and substantially extends the range of real-world business requirements that can be modelled. We have designed an FPT algorithm for WSP with

class-independent constraints. Our computational results demonstrate that our FPT algorithm is useful in practice for WSP with class-independent constraints, in particular for WSP instances that are too hard for SAT4J.

The full generalization of our approach is briefly described in Appendix C of [8] and the time complexity of the corresponding algorithm, $O^*(2^{rk \log_2 k})$ (r is the number of nested equivalence relations including $=$), indicates that it will remain practical at least when three rather than two equivalence relations are considered.

Acknowledgement. This research was supported by an EPSRC grant EP/K005162/1. The FPT algorithm's executable code and experimental data set are publicly available [10].

References

1 American National Standards Institute. *ANSI INCITS 359-2004 for Role Based Access Control*, 2004.
2 D. A. Basin, S. J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security and business objectives. *J. Comput. Security*, 22(5):661–698, 2014.
3 D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society, 1989.
4 D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Engineering algorithms for workflow satisfiability problem with user-independent constraints. In J. Chen, J.E. Hopcroft, and J. Wang, editors, *Frontiers in Algorithmics, FAW 2014*, volume 8497 of *Lecture Notes in Computer Science*, pages 48–59. Springer, 2014.
5 D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Iterative plan construction for the workflow satisfiability problem. *J. Artif. Intel. Res.*, 51:555–577, 2014.
6 D. Cohen, J. Crampton, A. Gagarin, G. Gutin, and M. Jones. Algorithms for the workflow satisfiability problem engineered for counting constraints. *J. Comb. Optim.*, to appear, 2015. (DOI: 10.1007/s10878-015-9877-7).
7 J. Crampton. A reference monitor for workflow systems with constrained task execution. In E. Ferrari and G.-J. Ahn, editors, *SACMAT*, pages 38–47. ACM, 2005.
8 J. Crampton, A. V. Gagarin, G. Gutin, and M. Jones. On the workflow satisfiability problem with class-independent constraints. *CoRR*, abs/1504.03561, 2015.
9 J. Crampton, G. Gutin, and A. Yeo. On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Trans. Inf. Syst. Secur.*, 16(1):4, 2013.
10 A. Gagarin, J. Crampton, G. Gutin, and M. Jones. Implementation of the pattern-backtracking FPT algorithm and experimental data set for the WSP with class-independent constraints. `http://dx.doi.org/10.6084/m9.figshare.1502692`, Aug 2015.
11 D. Karapetyan, A. Gagarin, and G. Gutin. Pattern backtracking algorithm for the workflow satisfiability problem. In *Frontiers in Algorithmics 2015*, volume 9130 of *Lect. Notes Comput. Sci.*, pages 138–149. Springer, 2015.
12 D. Le Berre and A. Parrain. The SAT4J library, release 2.2. *J. Satisf. Bool. Model. Comput.*, 7:59–64, 2010.
13 W. Myrvold and W. Kocay. Errors in graph embedding algorithms. *J. Comput. Syst. Sci.*, 77(2):430–438, 2011.
14 Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.*, 13(4):40, 2010.