Verification of security sensitive workflows with data

Author: Camilo Andrés Núñez

Supervisors: Clara Bertolissi and Pierre-Alain Reynier

Master 2 Informatique et Mathématiques Discrètes
Faculty of Sciences
Aix-Marseille Université
France
June 2024

Abstract:

This internship had three main objectives. First, using the approach presented in [3], we will formalize Security-Sensitive Data Aware Workflows (SS-DAWs). Second, we grant soundness, completeness and decidability of the backward reachability technique for testing satisfiability of SS-DAWs parametrized on users and, more generally, the read-only database. In this context, soundness means that whenever the procedure terminates, the returned answer is correct; completeness means that if the workflow is satisfiable, the procedure will always discover it; and decidability refers to the termination of the process, even if the workflow is unsatisfiable. Third, we formally present an example that considers a great variety of constraints induced by data. Additionally, we provide the specifications of our running example for the Model Checker MCMT [7].

Contents

1 Introduction

This research work focuses on operational processes like workflows and their verification [8]. A workflow outlines a series of tasks initiated by humans or software agents acting on their behalf, along with constraints on the order of these tasks. Workflows are structured, repeatable sets of tasks aimed at achieving specific goals, such as delivering a service or product. From a security standpoint, access control constraints must also be considerate. Authorization policies ensure that only users with the necessary permissions perform certain tasks; for example, a bank teller may create a loan request, but only a manager can approve it. Additional authorization constraints, such as Separation of Duties (SoD) or Binding of Duties (BoD), require either different users or the same user to perform two tasks, respectively. We will employ an array-based specification method to model workflows [6]. Workflow verification will be conducted using model checking via Satisfiability-Modulo-Theories (SMT) techniques. For instance, in [1], the satisfiability of workflows with security constraints (security-sensitive workflows, SSW) is explored, along with a methodology for real-time workflow monitoring. In [3], authors examines the verification of safety properties in workflows that include data (data-aware workflows, DAW).TRE

The main objective in this work is, roughly speaking, to increment the results in [1] by using the results in [3]. [1] describes a real-time workflow monitor that permits, for instance, employees in a company, ask for permission to start a task that, in case all the previous necessary task are marked as executed and the user accomplishes the security constraints, the monitor gives the permission to start the task. This tool is composed of two main components: The off-line phase and the on-line phase. The off-line phase consists on the production of a reachability graph, that is, given a workflow diagram, the reachability graph shows a path from an initial state to a final state. In order to provide a tool that permits all the possible executions, it is important that the reachability graph shows all the possible paths from an initial state to a final state. On the other hand, the on-line phase consists on a program that, based on the reachability graph, monitors the execution of the workflow by accepting or denying permissions to users to start the different tasks. It is important to remark that in this work tasks are considered as binary state variable, that is, either done or not done, where in particular no data is considered.

Conversely, [3] focuses, among other objectives, on the verification of more complex workflows, called business processes, that notably involve a read-only and read-write database, where the verification process is done over all the possible instances of the read-only database, or in terms of [4], parametrized on the read-only database.

Returning to the main objective of this work, the aim is to be able to automatize the creation of a reachability graph, that at the same time is able to parametrize users like in [1], and handle data in the sense that considers all the possible new reachability paths that different configurations of the data may create. To this end, we need, in the theoretical aspect, to define an approach to model our SS-DAWF, define a process that executes the desired search, and

guarantee the correction and termination of such process. On the more practical aspect, we make use of a tool that permits us input the specification of a workflow, and returns the reachability graph. Regarding the second aspect, the tool we have chosen to use is MCMT [7], since it was the tool used in both of the main articles in which this work is based.

Finally, we decided to devote section 5 to give a brief explanation on how workflow specifications should be done with the tool MCMT. In this section we provide a concise description of the specifications of our running example, which considers a great variety of constraints induced by data. The file with the specifications to be run with the tool can be found in the attached folder.

2 Preliminaries

We proceed to give basic definitions about workflows taken from [9]. Let $\mathcal{U}$ be an unbounded set of user identifiers.

Definition 2.1. *A **workflow (WF)** is as a tuple $\langle S, \preceq \rangle$, where S is a set of tasks and $\preceq \subseteq S \times S$ defines a partial order among tasks in S.*

Intuitively, in a workflow $\langle S, \preceq \rangle$, $s_i \preceq s_j (i \neq j)$ indicates that task s_i must be performed before task s_j. Tasks s_i and s_j may be performed concurrently if neither $s_i \preceq s_j$ nor $s_j \preceq s_i$.

Definition 2.2. *A **security-sensitive workflow (SS-WF)** is a tuple $\langle S, \preceq, \mathcal{C} \rangle$, where the set of constrains $\mathcal{C} = (\rho_1, \rho_2)$, with ρ_1, ρ_2 relations over S, is such that:*

- *If $(s_1, s_2) \in \rho_1$, the user who performs s_1 must be the same user who performs s_2.*
- *If $(s_1, s_2) \in \rho_2$, taks s_1 and s_2 must be performed by different users.*

Note that WFs can be viewed as SS-WFs where $\rho_1, \rho_2 = \emptyset$. So the following definitions apply for both cases.

Definition 2.3. *A **plan** P for a SS-WF $W = \langle S, \preceq, C \rangle$ is a subset of $\mathcal{U} \times S$ such that, for every step $s \in S$, there is exactly one tuple (u, s) in P, where $u \in \mathcal{U}$.*

Definition 2.4. *We say that a plan P is **valid** for W if and only if for every $(u, s) \in P$, no constraint in C is violated. We say that W is **satisfiable** if and only if there exists a plan P that is valid for W.*

Note that there can be multiple valid plans for a SS-WF W. In fact, what they mean by parametrization of users in [1] is to find all the possible valid plans of a given workflow. Since the set of users $\mathcal{U}$ is unbounded, parametrization is done over the number of users, not on their identifiers.

To fix ideas, consider the following examples of a SS-WFs in BPM notation[1] in figures 1, 2, 3.

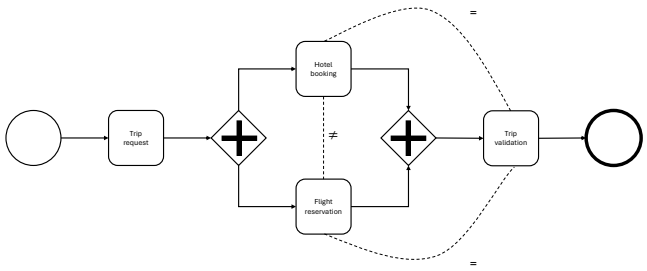


Figure 1: Example of an unsatisfiable workflow.

Since we are interested in verification involving data, we are forced to have a formalization for out database. For this aim, we will make use of some first order logic notions that we will present here.

We adopt the usual first-order syntactic notions of signature, term, atom, (ground) formula, and so on. We use $\underline{u}$ to represent a tuple $\langle u_1, \ldots, u_n \rangle$. Our signatures Σ are multi-sorted and include equality for every sort, which implies that variables are sorted as well. Depending on the context, we keep the sort of a variable implicit, or we indicate explicitly in a formula that variable x has sort S by employing notation $x : S$. The notation $t(\underline{x}), \phi(\underline{x})$ means that the term t, the formula ϕ has free variables included in the tuple $\underline{x}$. Constants and function symbols f have sources $\underline{S}$ and a target S', denoted as $f : \underline{S} \longrightarrow S'$. We assume that terms and formulae are well-typed, in the sense that the sorts of variables, constants, and relations, function sources/targets match. A formula is said to be universal (resp., existential) if it has the form $\forall \underline{x}(\phi(\underline{x}))$ (resp., $\exists \underline{x}(\phi(\underline{x}))$), where ϕ is a quantifier-free formula. Formulae with no free variables are called **sentences**. From the semantic side, we use the standard notions of a Σ-structure $\mathcal{M}$ and of truth of a formula in a Σ-structure under an assignment to the free variables. A Σ-theory T is a set of Σ-sentences; a **model** of T is a Σ-structure $\mathcal{M}$ where all sentences in T are true. We use the standard notation $T \models \phi$ to say that ϕ is true in all models of T for every assignment to the free variables of ϕ. We say that ϕ is T**-satisfiable** iff there is a model $\mathcal{M}$ of T and an assignment to the free variables of ϕ that make ϕ true in $\mathcal{M}$. In the following (cf. Section 3) we specify transitions of an artifact-centric system using first-order formulae. To obtain a more compact representation, we make use there of definable extensions as a means for introducing so-called case-defined functions. We fix a signature Σ and a Σ-theory T; a T-partition is a finite set $\kappa_1(\underline{x}), \ldots, \kappa_n(\underline{x})$ of quantifier-free formulae such that $T \models \forall \underline{x} \bigvee_{i=1}^{n} \kappa_i(\underline{x})$

[1]Full notation can be found in the official Bussiness Process Model and Notation website (https://www.bpmn.org/).

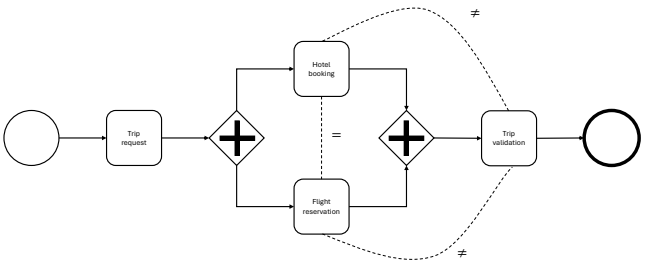


Figure 2: A similar workflow as in figure 1, but satisfiable since we changed the security constraints.

and $T \models \bigwedge_{i \neq j} \forall \underline{x}^{\neg} (\kappa_i(\underline{x}) \wedge \kappa_j(\underline{x}))$. Given such a T-partition $\kappa_1(\underline{x}), \ldots, \kappa_n(\underline{x})$ together with Σ-terms $t_1(\underline{x}), \ldots, t_n(\underline{x})$ (all of the same target sort), a case-definable extension is the Σ'-theory T', where $\Sigma' = \Sigma \cup \{F\}$, with F a "fresh" function symbol (i.e., $F \notin \Sigma$), and $T' = T \cup \bigcup_{i=1}^{n} \{\forall \underline{x} (\kappa_i(\underline{x}) \rightarrow F(\underline{x}) = t_i(\underline{x}))\}$. Intuitively, F represents a case-defined function, which can be reformulated using nested if-then-else expressions and can be written as $F(\underline{x}) :=$ case of $\{\kappa_1(\underline{x}) : t_1; \cdots ; \kappa_n(\underline{x}) : t_n\}$. By abuse of notation, we identify T with any of its case-definable extensions T'. In fact, it is easy to produce from a Σ'-formula ϕ' a Σ-formula ϕ equivalent to ϕ' in all models of T' : just remove (in the appropriate order) every occurrence $F(\underline{v})$ of the new symbol F in an atomic formula A, by replacing A with $\bigvee_{i=1}^{n} (\kappa_i(\underline{v}) \wedge A(t_i(\underline{v})))$[2].

3 Modeling workflows with data

Since our objective is to model data-aware security-sensitive workflows (DA SS-WF), we will use Relational Artifact Systems (RAS), following the approach in [3]. Intuitively, we want to be able to model workflows in which there is available a **read-only database (catalog)** that stores in one of its sheets the members of the company along with their relevant information[3]. Additionally, we need a **read-write database (repository)** to, for instance, store the users that have already executed a task previously during the execution of the WF.

It is important to remark that what we are doing here goes beyond verifying if a workflow is satisfiable for a fixed catalog. Our approach permits us to verify a process parametrically on the catalog. That is, finding all the possible instances

[2] $A(t_i(\underline{v}))$ reffers to the atomic formula A replacing all the occurrences of $F(v)$ by $t_i(\underline{v})$.

[3] Note that we can have multiple sheets. For example, we can also have a sheet for the products that the company sells, and another for the clients.

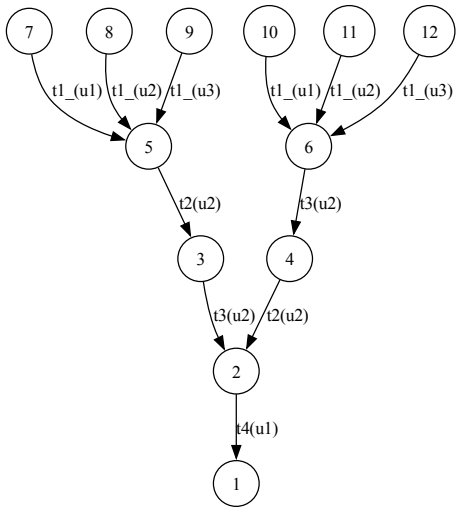


Figure 3: Parametrization of users in the satisfiable version of *Trip request*.

of catalogs for which the workflow is satisfiable. Clearly, synthesizing such instances exhaustively is not possible since there may be an infinite amount of favorable cases, so this is a way in which we leverage from a symbolic approach.

Artifacts will be formalized in the spirit of array-based systems. As described in [3],

> *"Array-based systems" is an umbrella term generically referring to infinite-state transition systems implicitly specified using a declarative, logic-based formalism. The formalism captures transitions manipulating arrays via logical formulae, and its precise definition depends on the specific application of interest. The overall state of the system is typically described by means of arrays indexed by process identifiers, and used to store the content of process variables like locations and clocks. These arrays are genuine second order function variables: they map indexes to elements, in a way that changes as the system evolves.*

3.1 Read-only database (catalog)

Definition 3.1. *A **catalog schema** is a pair $\langle \Sigma, T \rangle$, where:*

1. *Σ is a catalog signature, that is, a finite multi-sorted signature whose only symbols are equality, unary functions, and constants.*
2. *T is a catalog theory, that is, a set of universal Σ-sentences.*

Next, we refer to a catalog schema simply through its signature Σ and theory T, and denote by Σ_{srt} the set of sorts and by Σ_{fun} the set of functions in Σ. Since Σ contains only unary function symbols and equality, all atomic Σ-formulae are of the form $t_1(v_1) = t_2(v_2)$, where t_1, t_2 are possibly complex terms, and v_1, v_2 are either variables or constants.

We associate to a catalog signature Σ a **characteristic graph** $G(\Sigma)$ capturing the dependencies induced by functions over sorts. Specifically, $G(\Sigma)$ is an edge-labeled graph whose set of nodes is Σ_{srt}, and with a labeled edge $S \xrightarrow{f} S'$ for each $f : S \longrightarrow S'$ in Σ_{fun}. We say that Σ is **acyclic** if $G(\Sigma)$ is so. The leaves of Σ are the nodes of $G(\Sigma)$ without outgoing edges. These terminal sorts are called **value sorts**. Non-value sorts are called **id sorts**, and are conceptually used to represent (identifiers of) different kinds of objects. Value sorts, instead, represent datatypes such as strings, numbers, clock values, etc. We denote the set of id sorts in Σ by Σ_{ids}, and that of value sorts by Σ_{val}, hence $\Sigma_{srt} = \Sigma_{ids} \uplus \Sigma_{val}$.

Definition 3.2. *A **catalog instance** of a catalog schema $\langle \Sigma, T \rangle$ is a Σ-structure $\mathcal{M}$ that is a model of T and such that every id sort of Σ is interpreted in $\mathcal{M}$ on a finite set.*

Contrast this to arbitrary models of T, where no finiteness assumption is made. What may appear as not customary in Definition 3.2 is the fact that value sorts can be interpreted on infinite sets. This allows us, at once, to reconstruct the classical notion of DB instance as a finite model (since only finitely many values can be pointed from id sorts using functions), at the same time supplying a potentially infinite set of fresh values to be dynamically introduced in the working memory during the evolution of the artifact system.

We respectively denote by $S^{\mathcal{M}}$, $f^{\mathcal{M}}$, and $c^{\mathcal{M}}$ the interpretation in $\mathcal{M}$ of the sort S (this is a set), of the function symbol f (this is a set-theoretic function), and of the constant c (this is an element of the interpretation of the corresponding sort). Obviously, $f^{\mathcal{M}}$ and $c^{\mathcal{M}}$ must match the sorts in Σ. E.g., if f has source S and target U, then $f^{\mathcal{M}}$ has domain $S^{\mathcal{M}}$ and range $U^{\mathcal{M}}$.

We close the formalization of catalog schemas by discussing catalog theories, whose role is to encode background axioms. We illustrate a typical background axiom, required to handle the presence of undefined identifiers/values in the different sorts. To specify an undefined value we add to every sort S of Σ a constant $NULL\_S$ (written from now on, by abuse of notation, just as $NULL$, used also to indicate a tuple). Then, for each function symbol f of Σ, we add the following axiom to the catalog theory:

$$\forall x(x = NULL \leftrightarrow f(x) = NULL) \tag{1}$$

This axiom states that the application of f to the undefined value produces an undefined value, and it is the only situation for which f is undefined.

Remark 1. *In the SS-DAWFs that we intend to capture, the catalog theory consists of Axioms (1) only.*

3.2 Formal Properties of catalog Schemas

In this section we will introduce the background necessary to understand why we can freely make use of quantifier elimination in the main algorithm that will be introduced in section 4. It is, however, out of the scope of this work to present the proofs of the propositions stated in this section. The interested reader is referred to [2].

The theory T from definition 3.1 must satisfy a few crucial requirements for our approach to work. In this section, we define such requirements and show that they are matched.

A Σ-formula ϕ is a **Σ-constraint (or just a constraint)** if and only if it is a conjunction of literals. The **constraint satisfiability problem** for T asks: given an existential formula $\exists \underline{y}\phi(\underline{x}, \underline{y})$ (with ϕ a constraint), are there a model $\mathcal{M}$ of T and an assignment α to the free variables $\underline{x}$ such that $\mathcal{M}, \alpha \models \exists \underline{y}\phi(\underline{x}, \underline{y})$?

We say that T has the **finite model property (for constraint satisfiability)** if and only if every constraint ϕ that is satisfiable in a model of T is satisfiable in a catalog instance of T. The following propositions are stated without a proof, they can be found in [3].

Proposition 3.3. *The finite model property implies decidability of the constraint satisfiability problem in case T is recursively axiomatized.*

Proposition 3.4. *T has the finite model property in case Σ is acyclic.*

Definition 3.5. *A Σ-theory T has **quantifier elimination** iff for every Σ-formula $\phi(\underline{x})$ there is a quantifier-free formula $\phi'(\underline{x})$ such that $T \models \phi(\underline{x}) \leftrightarrow \phi'(\underline{x})$.*

It is known that quantifier elimination holds if quantifiers can be eliminated from primitive formulae, i.e., formulae of the kind $\exists \underline{y}\phi(\underline{x}, \underline{y})$, with ϕ a constraint. We assume that when quantifier elimination is considered, there is an effective procedure that eliminates quantifiers.

A catalog theory T does not necessarily have quantifier elimination; it is however often possible to strengthen T in a conservative way (with respect to constraint satisfiability) and get quantifier elimination.

Definition 3.6. *We say that T has a* **model completion** *iff there is a stronger theory $T^* \supseteq T$ (still within the same signature Σ of T) such that:*

i. Every Σ-constraint satisfiable in a model of T is also so in a model of T^*;

ii. T^* has quantifier elimination. T^* is called a model completion of T.

Proposition 3.7. *T has a model completion in case it is axiomatized by universal one-variable formulae and Σ is acyclic.*

Hereafter, we make the following assumption:

Assumption 3.8. *The catalog theories we consider have decidable constraint satisfiability problem, finite model property, and admit a model completion.*

This assumption is matched, for instance, in the following three cases:

i. When T is empty;

ii. When T is axiomatized by Axioms (1);

iii. When Σ is acyclic and T is axiomatized by finitely many universal one-variable formulae.

Remark 2. *Regarding the last case of the previous list, we can in particular add a special sort Int to T to define simple arithmetical operations such as "Add 30". Intuitively, what we do is defining arithmetic between 0 and 100 exhaustively, and assign $NULL\_Int$ to any operation resulting in a value before or beyond this range. This functions would create a cycle in the characteristic graph of Σ, which do not however harm the desired properties. Finite model property is preserved due to the fact that the sort Int has only one model up to isomorphism which is finite, and quantifier elimination can still be applied by using classic quantifier elimination techniques over integers such as Fourier-Motzkin.*

3.3 Read-write database (repository)

We are now in the position to define our formal model of Relational Artifact Systems (RASs), and to study parameterized safety problems over RASs. Since RASs are array-based systems, we start by recalling the intuition behind them.

In general terms, an array-based system is described using a multi-sorted theory that contains two types of sorts, one accounting for the indexes of arrays, and the other for the elements stored therein. Since the content of an array changes over time, it is referred to using a second-order function variable, whose interpretation in a state is that of a total function mapping indexes to elements (so that applying the function to an index denotes the classical read operation for arrays). The definition of an array-based system with array state variable a always requires: a formula $I(a)$ describing the initial configuration of the array a, and a formula $\tau\,(a, a')$ describing a transition that transforms the content of the array from a to a'. In such a setting, verifying whether the system can reach unsafe configurations described by a formula $K(a)$ amounts to check whether the formula $I\,(a_0) \wedge \tau\,(a_0, a_1) \wedge \cdots \wedge \tau\,(a_{n-1}, a_n) \wedge K\,(a_n)$ is satisfiable for some n. Next, we make these ideas formally precise by grounding array-based systems in the artifact-centric setting.

The RAS Formal Model. Following the tradition of artifact-centric systems, a RAS consists of a catalog, a ***read-write working memory (repository)*** for artifacts, and a finite set of actions that inspect the relational database and the working memory, and determine the new configuration of the working memory. In a RAS, the working memory consists of higher order variables. These variables (usually called arrays) are supposed to model evolving relations, that we will refer as ***artifact relations***. The idea is to treat artifact relations in a uniform way as we did for catalog: we need extra sort symbols and extra unary function symbols, the latter being treated as second-order variables.

Given a catalog Σ, an artifact extension of Σ is a signature Σ_{ext} obtained from Σ by adding to it some extra sort symbols[4]. These new sorts (usually indicated with letters $E, F, \ldots$) are called artifact sorts (or artifact relations by some abuse of terminology), while the old sorts from Σ are called basic sorts. In RAS, artifact and basic sorts correspond, respectively, to the index and the elements sorts mentioned in the literature on array-based systems. Below, given $\langle \Sigma, T \rangle$ and an artifact extension Σ_{ext} of Σ, when we speak of a Σ_{ext}- model of T, a catalog instance of $\langle \Sigma_{ext}, T \rangle$, or a Σ_{ext}-model of T^* , we mean a Σ_{ext}-structure $\mathcal{M}$ whose reduct to Σ respectively is a model of T, a catalog instance of $\langle \Sigma, T \rangle$, or a model of T^*.

An **artifact setting over** Σ_{ext} is a pair $(\underline{x}, \underline{a})$ given by a finite set $\underline{x}$ of individual variables and a finite set $\underline{a}$ of unary function variables: the latter are required to have an artifact sort as source sort and a basic sort as target sort. Variables in $\underline{x}$ are called **artifact variables**, and variables in $\underline{a}$ **artifact components**. Given a catalog instance $\mathcal{M}$ of Σ_{ext} , an **assignment to an artifact setting** $(\underline{x}, \underline{a})$ **over** Σ_{ext} is a map α assigning to every artifact variable $x_i \in \underline{x}$ of sort S_i an element $x^\alpha \in S_i^{\mathcal{M}}$ and to every artifact component $a_j : E_j \longrightarrow U_j$ (with $a_j \in \underline{a}$) a set-theoretic function $a_j^\alpha : E_j^{\mathcal{M}} \longrightarrow U_j^{\mathcal{M}}$. In RAS, artifact components and artifact variables correspond, respectively, to arrays and constant arrays (i.e., arrays with all equal elements) mentioned in the literature on array-based systems.

We can view an assignment to an artifact setting $(\underline{x}, \underline{a})$ as a catalog instance extending the catalog instance $\mathcal{M}$ as follows. Let all the artifact components in $(\underline{x}, \underline{a})$ having source E be $a_{i_1} : E \longrightarrow S_1, \cdots, a_{i_n} : E \longrightarrow S_n$. Viewed as a relation in the artifact assignment $(\mathcal{M}, \alpha)$, the artifact relation E "consists" of the set of tuples $\left\{ \left\langle e, a_{i_1}^\alpha(e), \ldots, a_{i_n}^\alpha(e) \right\rangle \mid e \in E^{\mathcal{M}} \right\}$. Thus each element of E is formed by an "entry" $e \in E^{\mathcal{M}}$ (uniquely identifying the tuple) and by "data" $\underline{a}_i^\alpha(e)$ taken from the read-only database $\mathcal{M}$. When the system evolves, the set $E^{\mathcal{M}}$ of entries remains fixed, whereas the components $\underline{a}_i^\alpha(e)$ may change.

In order to introduce verification problems in the symbolic setting of array-based systems, one first has to specify which formulae are used to represent sets of states, the system initializations, and system evolution. To introduce RASs we discuss the kind of formulae we use. In such formulae, we use notations like $\phi(\underline{x}, \underline{a})$ to mean that ϕ is a formula whose free constant array variables are among the $\underline{x}$ and whose free array variables are among the $\underline{a}$. Let $(\underline{x}, \underline{a})$ be an artifact setting over Σ_{ext}, where $\underline{x} = x_1, \ldots, x_n$ are the artifact variables and $\underline{a} = a_1, \ldots, a_m$ are the artifact components (their source and target sorts are left implicit).

An **initial formula** is a formula $\iota(\underline{x})$ of the form

$$\left(\bigwedge_{i=1}^{n} x_i = c_i \right) \wedge \left(\bigwedge_{j=1}^{m} \forall y. a_j(y) = d_j \right) \tag{2}$$

Where c_i, d_j are constants from Σ (typically, c_i and d_j are $undef$). A **state**

[4]We are assuming that all of our signatures have equality.

formula has the form $\exists \underline{e}\phi(\underline{e}, \underline{x}, \underline{a})$, where ϕ is quantifier-free and the $\underline{e}$ are individual variables of artifact sorts. A transition formula $\hat{\tau}$ has the form

$$\exists \underline{e} \left(\gamma(\underline{e}, \underline{x}, \underline{a}) \wedge \bigwedge_i x_i' = F_i(\underline{e}, \underline{x}, \underline{a}) \wedge \bigwedge_j \forall y. a_j' = G_j(y, \underline{e}, \underline{x}, \underline{a}) \right) \tag{3}$$

where the $\underline{e}$ are individual variables (of both basic and artifact sorts), γ (the 'guard') is quantifier-free, $\underline{x}', \underline{a}'$ are renamed copies of $\underline{x}, \underline{a}$, and the F_i, G_j (the 'updates') are case-defined functions.

Definition 3.9. *A Relational Artifact System (RAS) is a tuple*

$$\mathcal{S} = \langle \Sigma, T, \Sigma_{ext}, \underline{x}, \underline{a}, \iota(\underline{x}, \underline{a}), \tau\,(\underline{x}, \underline{a}, \underline{x}', \underline{a}') \rangle$$

where:

i. $\langle \Sigma, T \rangle$ *is a catalog schema.*

ii. Σ_{ext} *is an artifact extension of* Σ.

iii. $(\underline{x}, \underline{a})$ *is an artifact setting over* Σ_{ext} .

iv. ι *is an intitial formula.*

v. τ *is a disjunction of transition formulae.*

To fix ideas, consider the following version of *Trip request.*

Example 3.10. *We will present the workflow in figure 4 as a RAS. First, following the same approach as [1], we portray the workflow as a petri net in figure 5.*

Then, we proceed to represent formally our catalog, which consists, in one side, on a list of members of the company along with their status (intern or permanent), and on another, a list of the clients that have submitted a trip request.

$$\Sigma \text{ is } \Sigma_{srt} \cup \Sigma_{fun} \cup \{=\}$$

Where

$$\Sigma_{srt} \text{ is } \{UserID, Status, Order, Int\}$$

And

$$\Sigma_{fun} \text{ is } \left\{ \begin{array}{c} NULL\_Order : Order, 2-3 : Order, 3-2 : Order, \\ NULL\_Status : Status, intern : Status, \\ permanent : Status, NULL\_UserID : UserID, \\ NULL\_Status : Status, \\ userstat : UserID \to Status, +20 : Int \to Int \\ +30 : Int \to Int, +60 : Int \to Int \end{array} \right\}.$$

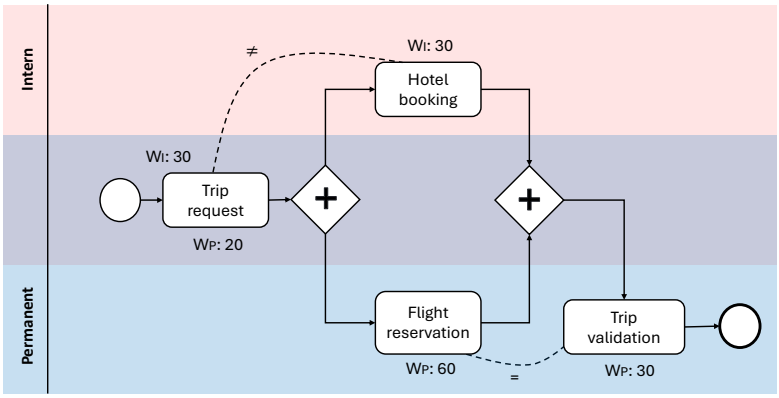


Figure 4: Workflow *Trip request*. The W's represent the workloads associated to each task depending of the status of the worker. No user can surpass his maximum capacity, which here is 100. This should be interpreted as the working hours of the user, which naturally vary depending on whether the user is permanent or an intern.

As stated in remark (1), our catalog theory will always contain axioms (1)[5]*. For readability, we omitted all the axioms and constants required to define the restricted arithmetic we use here.*

$$T = \{\forall x(x = NULL\_UserID \leftrightarrow userstat(x) = NULL\_Status).\}$$

Now we proceed to define our artifact extension:

$$\Sigma_{ext} = \{userIndex, clientIndex\},$$

and our artifact setting $(\underline{x}, \underline{a})$*, where*

$$\underline{x} = (flag, p1, p2, p3, p4, p5, p6, dt1, dt2, dt3, dt4),$$

and

$$\underline{a} = (user, ht1, ht2, ht3, ht4, workload).$$

Then, we present the initial and final state formulas:

$$\iota :\equiv \quad \forall i : userIndex \left(\begin{array}{c} p0 = true \wedge p1 = false \wedge p2 = false \wedge p3 = false \\ \wedge p4 = false \wedge p5 = false \wedge p6 = false \wedge dt1 = false \\ \wedge dt2 = false \wedge dt3 = false \wedge dt4 = false \wedge ht1[i] = false \\ \wedge ht2[i] = false \wedge ht3[i] = false \wedge ht4[i] = false \end{array} \right)$$

[5]For readability, we call the *undef* value of each sort $NULL\_ <$ name of the sort $>$.

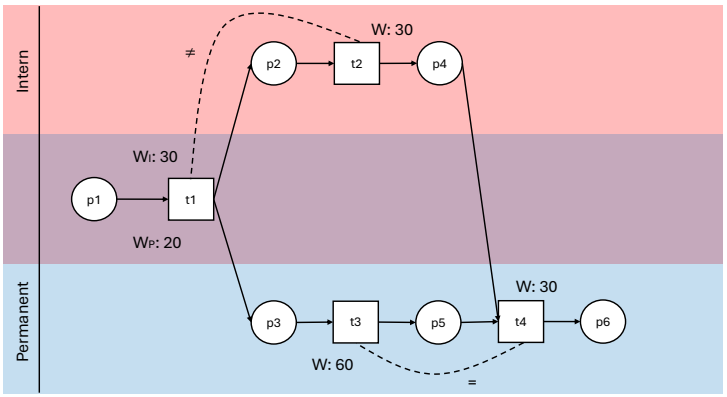


Figure 5: *Trip request* portrayed as a Petri net. In particular, a task needs a token in the place behind in order to be executed, and when its executed it places the token in front of it.

$$v :\equiv \quad \exists i : userIndex \left(\begin{array}{c} flag \neq NULL\_Order \wedge p0 = true \wedge p1 = false \wedge p2 = false \\ \wedge p3 = false \wedge p4 = false \wedge p5 = false \wedge p6 = false \\ \wedge dt1 = false \wedge dt2 = false \wedge dt3 = false \wedge dt4 = false \end{array} \right)$$

Finally, we present the transitions. We proceed to make a transition for each task in the workflow. In the case of the first transition, we just have to check that the artifact index we are going to use to assign task one, has a user assigned, regardless weather his status is permanent or intern.

$$\tau_1 :\equiv \quad \exists i : userIndex, d : UserID, k : clientIndex \left(\begin{array}{c} p1 = true \wedge p2 = false \wedge p3 = false \wedge p4 = false \wedge p5 = false \\ \wedge p6 = false \wedge user[i] = d \wedge d \neq NULL\_UserID \\ \wedge userstat[d] = intern \wedge p1' = false \wedge p2' = true \wedge \\ p3' = true \wedge dt1' = true \wedge client[k] \neq NULL\_ClientID \\ \wedge \forall j.ht1'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j.workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +30(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

$$\tau_1' :\equiv \quad \exists i : userIndex, d : UserID$$
$$\left(\begin{array}{c} p1 = true \wedge p2 = false \wedge p3 = false \wedge p4 = false \wedge p5 = false \\ \wedge p6 = false \wedge user[i] = d \wedge d \neq NULL\_UserID \\ \wedge userstat[d] = permanent \wedge p1' = false \wedge p2' = true \wedge \\ p3' = true \wedge dt1' = true \wedge client[k] \neq NULL\_ClientID \\ \wedge \forall j. ht1'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j. workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +20(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

Remark 3. There is a particular thing about tasks two and three, they can be done concurrently. If our objective was only to check if our system is unsafe against a given undesired property, we would just create a transition per task and, for example, in the case of task two, we would set the guard to grant artifact variable $p2$ to be true and not to put any constraint on $p3$, since we are admitting task two to be done before or after task three. However, since our interest is to show all the possible reachability paths, this approach does not work since after considering the first possible order in which task one and two can be executed, the other way is considered redundancy and is not included in the reachability graph due to the satisfiability test in line 4 of algorithm 2. More comments will be done on this, but for now we just state that we are forced to do a transition for all the possible configurations of the Petri net, and it is here where we will make use of the artifact variable $flag$.

$$\tau_2 :\equiv \quad \exists i : userIndex, d : UserID$$
$$\left(\begin{array}{c} p1 = false \wedge p2 = true \wedge p3 = true \wedge p4 = false \wedge p5 = false \\ \wedge p6 = false \wedge flag = NULL\_Order \wedge flag' = 2 - 3 \wedge \\ user[i] = d \wedge userstat[d] = intern \wedge p2' = false \wedge p4' = true \\ \wedge \forall j. ht2'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j. workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +30(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

$$\tau_2' :\equiv \quad \exists i : userIndex, d : UserID$$
$$\left(\begin{array}{c} p1 = false \wedge p2 = true \wedge p3 = false \wedge p4 = false \wedge p5 = true \\ \wedge p6 = false \wedge flag = 3 - 2 \wedge user[i] = d \\ \wedge userstat[d] = intern \wedge p2' = false \wedge p4' = true \\ \wedge \forall j. ht2'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j. workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +30(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

$$\tau_3 :\equiv \quad \exists i : userIndex, d : UserID$$
$$\left(\begin{array}{c} p1 = false \wedge p2 = true \wedge p3 = true \wedge p4 = false \wedge p5 = false \\ \wedge p6 = false \wedge flag = NULL\_Order \wedge flag' = 3 - 2 \wedge user[i] = d \\ \wedge userstat[d] = permanent \wedge p3' = false \wedge p5' = true \\ \wedge \forall j.ht2'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j.workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +60(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

$$\tau_3' :\equiv \quad \exists i : userIndex, d : UserID$$
$$\left(\begin{array}{c} p1 = false \wedge p2 = false \wedge p3 = true \wedge p4 = true \wedge p5 = false \\ \wedge p6 = false \wedge flag = 2 - 3 \wedge user[i] = d \\ \wedge userstat[d] = permanent \wedge p3' = false \wedge p5' = true \\ \wedge \forall j.ht2'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j.workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +60(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

Finally, we specify the transition for task 4.

$$\tau_4 :\equiv \quad \exists i : userIndex, d : UserID$$
$$\left(\begin{array}{c} p1 = false \wedge p2 = false \wedge p3 = false \wedge p4 = true \wedge p5 = true \\ \wedge p6 = false \wedge user[i] = d \wedge userstat[d] = permanent \\ \wedge p4' = false \wedge p5' = false \wedge p6' = true \\ \wedge \forall j.ht2'[j] = \left(\begin{array}{l} if\ j = i\ then\ true \\ else\ ht1[j] \end{array} \right) \\ \wedge \forall j.workload'[j] = \left(\begin{array}{l} if\ j = i\ then\ +30(workload[j]) \\ else\ workload[j] \end{array} \right) \end{array} \right)$$

3.4 Backward Reachability

A safety formula for $\mathcal{S}$ is a state formula v describing undesired states of $\mathcal{S}$. As usual in array-based systems, we say that $\mathcal{S}$ is safe with respect to v if intuitively the system has no finite run leading from ι to v. Formally, there is no catalog instance $\mathcal{M}$ of $\langle \Sigma_{ext}, T \rangle$, no $k \geq 0$, and no assignment in $\mathcal{M}$ to the array variables $\underline{x}^0, \underline{a}^0 \ldots, \underline{x}^k, \underline{a}^k$ such that the formula

$$\iota\left(\underline{x}^0, \underline{a}^0\right) \wedge \tau\left(\underline{x}^0, \underline{a}^0, \underline{x}^1, \underline{a}^1\right) \wedge \cdots \wedge \tau\left(\underline{x}^{k-1}, \underline{a}^{k-1}, \underline{x}^k, \underline{a}^k\right) \wedge v\left(\underline{x}^k, \underline{a}^k\right)$$

is true in $\mathcal{M}$ (here $\underline{x}^i, \underline{a}^i$ are renamed copies of $\underline{x}, \underline{a}$). The safety problem for $\mathcal{S}$ is the following: given a safety formula v decide whether $\mathcal{S}$ is safe with respect to v. Given our aims, our safety formula will be the state representing the completion of the workflow.

4 Soundness, completeness, and decidability

In this section we will present the algorithm we use to find all the possible paths to satisfy a SS-DAWF, which inherently gives us all the catalog instances that make the SS-DAWF satisfiable.

Before we present the algorithm, we must introduce the definition of preimage of a state formula.

Definition 4.1. *Given a RAS S and a state formula $\phi(\underline{z})$, we define the* ***preimage*** *of $\phi(\underline{z})$ as $Pre(\tau, \phi) = \bigvee_{\tau_i \in \tau} \exists \underline{z}'(\tau_i(\underline{z}, \underline{z}') \wedge \phi(\underline{z}'))$*[6].

We proceed now to present the standard Backward reachability algorithm. Our approach, even though it follows the same principles, strictly speaking a very different algorithm. In particular, our approach is greater in size. We decided to present the standard procedure because it is an easier task for a novice reader to understand the principles of backward reachability with it, thus making it an easier to understand algorithm 2, our approach.

Algorithm 1: Backward reachability algorithm

```
1 Function BReach(v):
2     φ ← v; B ←⊥;
3     while φ ∧ ¬B is T-satisfiable do
4         if ι ∧ φ is T-satisfiable then
5             return unsafe;
6         B ← φ ∨ B ;
7         φ ← Pre(τ, φ) ;
8         φ ← QE(T*, φ);
9     return (safe, B);
```

In algorithm 1 ϕ represents the states that are currently being explored, whereas B represents the set of explored states. The satisfiability test in line 3 is known in the literature as **fixpoint test**, and we require an SMT solver for it. This test permits us to check if the state ϕ has already been previously explored. In fact, $\phi \wedge \neg B$ is the negation of $\phi \Rightarrow B$, so if $\phi \wedge \neg B$ is not $T-$satisfiable, it means that $\phi \Rightarrow B$ is a tautology, which likewise implies that the state ϕ has already been explored or ϕ is itself not satisfiable in T. Whereas if $\phi \wedge \neg B$ is satisfiable, ϕ represents an unexplored state.

The satisfiability test in line 4 checks if ϕ is compatible with the initial state, for which if it's the case, we can grant that the system is unsafe.

In line 6 we just add ϕ to B, given that ϕ had previously passed the fixpoint test. Moving on to line 6, as mentioned in definition 4.1, we proceed now to present the actual definition of preimage with an example. Consider a simple

[6]The definition of preimage is actually slightly different from this one. We can think of this expression as notation for the actual definition which will be presented in the explanation of algorithm 1 below.

RAS consisting of only one array a taking boolean values, and consider the state formula

$$\phi \equiv \exists i_1 . a[i_1] = true \tag{4}$$

and the transition

$$\tau \equiv \exists i_2 . a[i_2] = NULL\_Bool \wedge \forall j . a'[j] = F(j, i_2) \tag{5}$$

$$\text{where } F(j, i_2) = \begin{array}{l} True \text{ if } j = i_2 \\ a[j] \text{ if } j \neq i_2. \end{array}$$

If we applied definition 4.1 to ϕ and τ, we can easily see that the formula $\exists . a'(\tau(a, a') \wedge \phi(a'))$ is not a state formula since we are quantifying the second order variable a'. However, even if we consider the formula $\tau(a, a') \wedge \phi(a')$, whose satisfiability is equivalent to $\exists . a'(\tau(a, a') \wedge \phi(a'))$, it would still not be a state formula since the sub-formula $\tau(a, a')$ contains a universal quantifier.

The procedure to compute the preimage is the following. Consider the formula $\tau(a, a') \wedge \phi(a')$, that is,

$$\exists i_1, i_2 . \forall j . a'[j] = F(j, i_2) \wedge a[i_2] = NULL\_Bool \wedge a'[i_1] = true.$$

Remove all the occurrences of the literal containing case-defined functions (the literals universally quantified), and replace all the appearances of the updated arrays with the case-defined functions that update them respectively. Which in the case of our example would be the formula

$$\exists i_1, i_2 . a[i_2] = NULL\_Bool \wedge F(i_1, i_2) = true$$

This is still not a state formula since case-defined functions they are not allowed to have case-defined functions. However, we can always find an equivalent state formula by looking at the definition of the case-defined functions. In our example it would be the state formula

$$\exists i_1, i_2 . a[i_2] = NULL\_Bool \wedge (a[i_1] = NULL\_Bool \vee a[i_1] = true). \tag{6}$$

Remark 4. *It is important to mention that the satisfiability modulo T^* of $Pre(\tau, \phi)$ and $\tau(\underline{z}, \underline{z'}) \wedge \phi(z')$ are equivalent.*

Coming back to algorithm 1, in line 7, the operation QE generates a quantifier free formula equivalent to ϕ modulo T^*. This formula is granted to exist by proposition 3.7 and and remark 2. Finally, we keep looking for new states through the preimages of the already explored states until we find a reachability path or there are no more new states, whatever happens first.

After understanding how standard backward reachability works, we are now able to present our approach, that not only verifies if our system is safe against a given property, but also builds a reachability graph.

Algorithm 2: Building a symbolic reachability graph

Input: ι, τ, v
Output: $RG = (N, \sigma, I)$

```
1  i ← 0; N.append(v); I ← [ ]; σ ← [ ];
2  SafetyCheck ← safe ; B ← ⊥ ;
3  while i < len(N) do
4      if ι ∧ N(i) is T-satisfiable then
5          SafetyCheck ← unsafe; I.append(N(i));
6      B ← N(i) ∨ B;
7      forall τ_i ∈ τ do
8          φ ← Pre(τ_i, N(i));
9          φ ← QE(φ);
10         forall γ ∈ split(φ) do
11             if γ ∧ ¬B is T-satisfiable then
12                 N.append(γ);
13                 σ ← (i, τ_i, position(QE(γ)));
14     i ← i + 1;
15 return (N, σ, I, SafetyCheck);
```

As mentioned before, algorithms 1 and 2 have a lot of subtle differences but share the same core idea. Furthermore, the main differences they have with respect to their output are 2. First, algorithm 1 does its search by doing a bulk update in the set of explorable states. That is, it does not considerate nodes. Second, it stops when it finds the first reachability path, whereas algorithm 2 only stops if it has already explored all the set of reachable states.

In algorithm 2, N is the set of nodes, σ the directed segments, and I the set of initial states. The function split represents an effective procedure to transform a preimage into an equivalent disjunctive normal form in which every maximal conjunction of literals have a different set of quantified variables of artifact sort, who in turn are always differentiated[7]. To clarify concepts, again using the simple RAS we used in the explanation of algorithm 1, if we apply split to the preimage formula 6, we obtain the state formula

$$\exists i_1(a[i_1] = NULL\_Bool) \vee \exists i_1, i_2((i_1 \neq i_2) \wedge a[i_2] = NULL\_Bool \wedge a[i_1] = true)$$

So one nodal state formula obtained as an extension of 4 under the transition 5 would be

$$\exists i_1(a[i_1] = NULL\_Bool)$$

And the other would be

[7] In our framework, the state formula representing a node of the reachability graph is always a maximal conjunction of literals from a disjunctive normal form representing the preimage of another node.

$$\exists i_1, i_2((i_1 \neq i_2) \wedge a[i_2] = NULL\_Bool \wedge a[i_1] = true)$$

However, it is worth noting that the second one implies the first one, so if they are explored in the same order, the second formula would pass the fixpoint test and would not be stored as a node in the final output.

Our next step is to provide a sketch of the proof of soundness, completeness and decidability. One of the advantages of algorithm 2 is that its output proofs its soundness.

Theorem 4.2. *Algorithm 2 is correct.*

Proof. The first implication is given by the output of our algorithm. In fact, We just need to consider a complete path from the reachability graph, and take the model of the first state of the path.

Conversely, suppose there exists $n \in \mathbb{N}$ such that

$$\mathcal{M} \models \iota(a_0) \wedge \tau(a_0, a_1) \wedge \cdots \wedge \tau(a_{n-1}, a_n) \wedge \upsilon(a_n).$$

Since $\tau = \bigvee_{i \in I} \tau_i$, we have that there are $i_1, \ldots, i_n$ such that

$$\mathcal{M} \models \iota(a_0) \wedge \tau_{i_1}(a_0, a_1) \wedge \cdots \wedge \tau_{i_n}(a_{n-1}, a_n) \wedge \upsilon(a_n)$$

And since our algorithm checks the preimage under every transition at every node, it would find this path and return $unsafe$.

□

Concerning completeness and decidability, it is clear that in the case of algorithm 2 they are equivalent, since it only stops until checking all the possible states of the system. Additionally, since definition 2.1 requires our workflows to be a partial order[8], completeness and decidability are granted. This is due to, on one side, the fact that the preimage of a node is always equivalent to the disjunction of a finite set of nodal formulae. And on the other, propositions x,y,z.

5 The tool MCMT

This section focuses on explaining how to write the specifications for describing workflows modelled as RASs using the tool MCMT. Furthermore, we discuss the limitations of using this tool for our objectives.

We explain how to write the specifications of our running example. The interested reader can find the complete documentation in the manual that comes with the download of the tool [7]. The objective of this section is rather to explain how the tool is used in the case of data-aware workflows as explained in [5], since the tool is used also to study different types of processes such as timed and fault-tolerant distributed systems, imperative programs, etc.

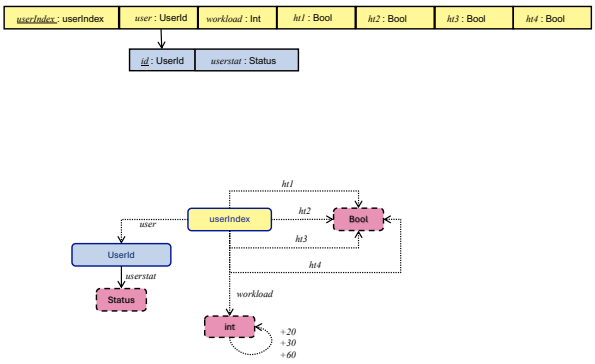


Figure 6: On the top: relational view of the catalog signature and the corresponding artifact relations; each cell denotes an attribute with its type, underlined attributes denote primary keys, and directed edges capture foreign keys; value sorts are shown in pink, basic id sorts in blue, and artifact id sorts in yellow. On the bottom: characteristic graph of the human Trip request catalog signature augmented with the signature of the artifact extension.

We proceed now to present in figure 6 the formalization of our SS-DAWF as a RAS.

We move on now to explain how it must be specified. In general, we first specify the catalog, then artifacts, followed by initial and final state formulae, and in finally the transitions. With respect to the catalog, the first part of the specification are the definition of its sorts.

:smt (define-type UserID)
:smt (define-type Status)
:smt (define-type Order)

Since the sorts int and value are already included in the tool there is no need to include them. Then we define function (except for addition) and constants.

:smt (define userstat ::(-> UserID Status))
:smt (define permanent :: Status)
:smt (define intern :: Status)
:smt (define TwoThree :: Order)
:smt (define ThreeTwo :: Order)

[8]That is, no cycles are allowed

To finish the specification of the catalog schema, we make a summary of the sorts, functions, and constants we will use during the verification process. This summary permits using the same catalog schema specification for different workflows.

:db_driven
:db_sorts UserID Status Order
:db_functions userstat
:db_constants permanent intern TwoThree ThreeTwo

Next, we move on to the specifications of the artifact setting. As it is shown in figure 6, there is just one artifact sort for the artifact components[9], and five functions, all of them with source sort $UserID$, one with image sort *int*, and the other with image of sort *Bool*. Contrarily to what we do with the catalog schema, we do not specify these functions. What we do is just define each artifact component separately as follows, and defining the transitions in such a way that related components update accordingly.

:global sat bool
:global flag Order
:global p0 bool
:global p1 bool
:global p2 bool
:global p3 bool
:global p4 bool
:global p5 bool
:global p6 bool
:global dt1 bool
:global dt2 bool
:global dt3 bool
:global dt4 bool
:local user UserID
:local workload int
:local ht1 bool
:local ht2 bool
:local ht3 bool
:local ht4 bool

As it can be inferred from the specification, artifact components are specified with the keyword *local*, whereas artifact variables are specified with the keyword *global*.

We will now move to the specification of the initial and final states formulae. The tool is already settled to receive a universal initial state formula and exis-

[9]There is also one additional artifact sort per artifact variable. However, in the literature they are usually not shown in the relational nor graphic representation of the RAS since they are typically numerous and their treatment is very simple since their sorts only require one index.

tential final state formula, so it is not necessary to use quantifiers. The initial formula is specified as:

:initial
:var x
:cnj (= sat true)(= p0 true)(= p1 false)(= p2 false)(= p3 false)(= p4 false)
(= p5 false)(= p6 false)(= dt1 false)(= dt2 false)(= dt3 false)(= dt4 false)
(= ht1[x] false)(= ht2[x] false)(= ht3[x] false)(= ht4[x] false)(= workload[x] 0)

And the final state formula as:

:max_unsafe_number 20
:u_cnj (sat true)(= p0 false)(= p1 false)(= p2 false)(= p3 false)(= p4 false)
(= p5 false)(= p6 true)(= dt1 true)(= dt2 true)(= dt3 true)(= dt4 true)

The maximum default number of literals in the final state formula is 10, this is why we use the command *:max_unsafe_number 20*.

Figure 5 shows how the catalog and repository we defined above look like. We can see that both have two different sheets, however their formalization is very different. In the case of the catalog, every column is corresponds to a different basic sort, and when two sorts appear in the same sheet it means that they share the same domain, the leftmost ID sort of the sheet. Regarding the repository, there is a sheet per artifact sort, and every column is a local array having as set of index the artifact sort in the leftmost column.

uIndex : userIndex	*user* : UserId	*userStat*: Status	*Workload*: int	*ht1* : Bool	*ht2* : Bool	*ht3*: Bool	*ht4*: Bool
1							
2							
3							
4							

.
.
.

idUser : UserId	*userstat* : Status
001	Intern
002	Permanent
003	Permanent
004	Intern

cIndex : clientIndex	*client* : ClientID
1	
2	
3	

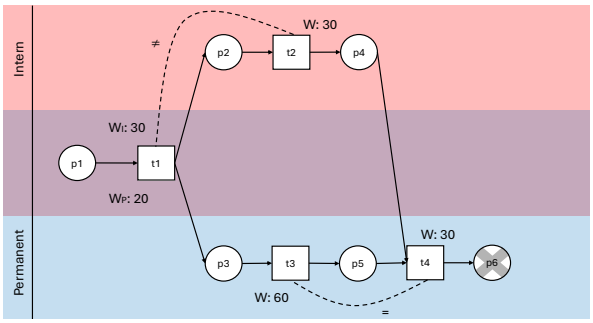
.
.
.

idClient : ClientID
01
02
03

Figure 7: An instance of the database in our running example. The yellow sheets represent the repository, and the blue sheets represent the catalog.

The specifications for the transitions are usually very lengthy since the tool requires to mention the update of every element in the artifact setting, even if it does not change at all. This is why we opted not to explain how they are done here, but to attach to this report the specifications of this workflow.

Let's see now how the repository, graph, and configurations of the workflow evolve during the backward reachability procedure[10].

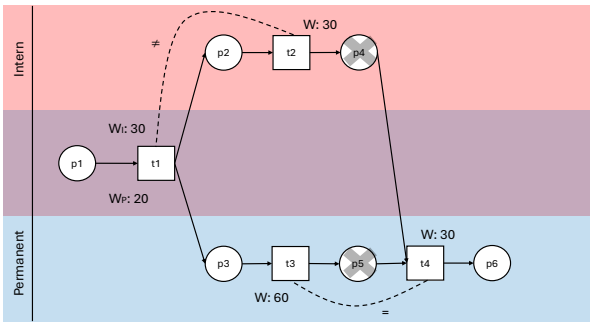


uIndex : userIndex	*user* : UserId	*ht1* : Bool	*ht2* : Bool	*ht3* : Bool	*ht4* : Bool	*ht5* : Bool	Workload: int
1							
2							
3							
4							

1

Figure 8: Step 1

[10] In fact, we present the evolution of the process when using a depth-first approach, whereas our algorithm is actually breadth-first, which makes the process considerably longer.



uIndex : userIndex	*user* : UserId	*userStat*: Status	*Workload*: int	*ht1* : Bool	*ht2* : Bool	*ht3*: Bool	*ht4*: Bool
1	001	Permanent	<=70			True	False
2							
3							
4							

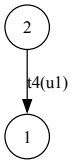
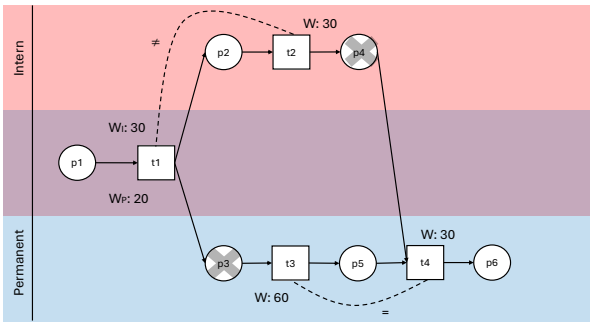


Figure 9: Step 2



uIndex : userIndex	*user* : UserId	*userStat*: Status	*Workload*: int	*ht1* : Bool	*ht2* : Bool	*ht3*: Bool	*ht4*: Bool
1	001	Permanent	<=10			False	False
2							
3							
4							

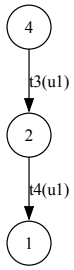
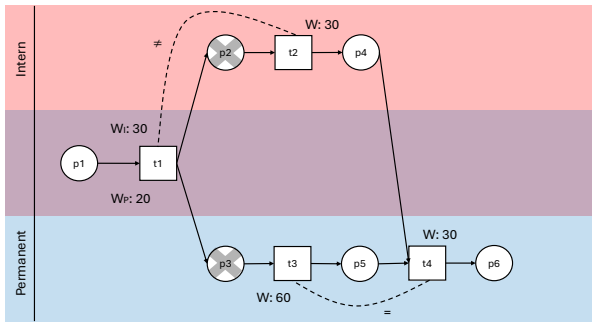


Figure 10: Step 3



uIndex : userIndex	*user* : UserId	*userStat*: Status	*Workload*: int	*ht1* : Bool	*ht2* : Bool	*ht3*: Bool	*ht4*: Bool
1	001	Permanent	<=10			False	False
2	002	Intern	<=70	False	False		
3							
4							

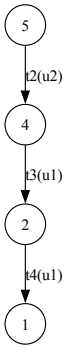
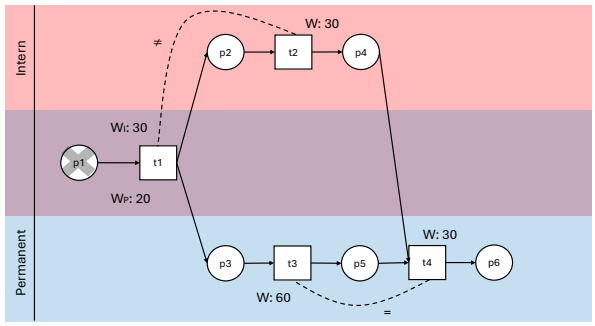


Figure 11: Step 4



uIndex : userIndex	*user* : UserId	*userStat*: Status	*Workload*: int	*ht1* : Bool	*ht2* : Bool	*ht3*: Bool	*ht4*: Bool
1	001	Permanent	<=10			False	False
2	002	Intern	<=70	False	False		
3	003	Permanent	<=80	False			
4							

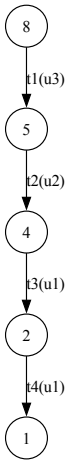


Figure 12: Step 5

Before passing to the conclusions, we make two more comments. First, the global variable *sat*, that did not appear in the initial version of *Trip request*, is used in order to simulate the execution of algorithm 2 in the tool, since the real algorithm that it is executing is algorithm 1. That is, the tool stops when it

finds the first reachability path, whereas in algorithm 2 we always explore all the possible states. This exhaustive exploration can be enforced by setting the variable sat to be $false$ in the initial formula.

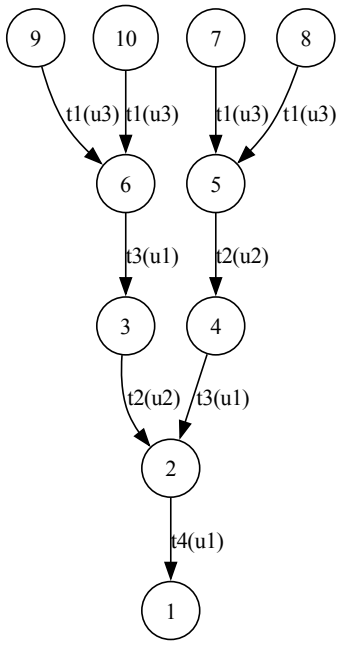


Figure 13: Reachability graph produced by the tool MCMT having as input the specifications of Trip request and activating the option $sat = false$.

Second, in the reachability graph shown in figure 13, the catalog for every path is constant. This can be understood from the fact that every time we apply the preimage in backward reachability we give more and more constraints to the catalog. Thus, the states that define nodes $7, 8, 9$ and 10 are the catalog instances for which our workflow is satisfiable. Also, the state formula representing each one of the initial nodes tell us exactly the information required in the catalog to be able to execute the workflow in the way the reachability path indicates it. For example, we can see in figure 14 the state formula of node 7, which is given by the report that the tool outputs. This formula tells us that the catalog must have at least one user of status *permanent* ($z1$), and at least two users of status *intern* ($z2$ and $z3$).

```
∃z1, z2, z3.          (and (ENTRY z1) (sat true) (ENTRY z2) (ENTRY z3)
            ((userstat user[z3]) intern) (= permanent (userstat user[z1] )) )
                  (= intern (userstat user[z2] )) (= flag ThreeTwo) )
                (= ht1[z3] false) (= ht2[z2] false) (= ht3[z1] false) )
                    (= ht4[z1] false) (= p2 false) (= p3 false) )
                 (= p4 false) (= p5 false) (= p6 false) (= p0 true) )
                (= p1 true) (<= workload[z1] 10) (<= 0 workload[z1]) )
```

Figure 14: State formula of node 7 in figure 14.

6 Conclusions

To summarize this research work, our first achievement was to formalize the notion of the SS-DAWF as a RAS. Second, we introduced algorithm 2, which executes the task of finding all the possible reachability paths of a workflow, or tell us when it is unsatisfiable. With respect to completeness and decidability, we proved that in the case of our approach they are equivalent, and that in the case of the workflows we are studying, it is always granted. Finally, we presented an example in full detail that considered multiple instances of constraints induced by data, in particular being able to handle quantitative aspects such as workloads, and attached a folder containing its specifications and the material that the tool outputs after running them.

For a future work, the next logic step would be to study the behavior of algorithm 2 when used to verify workflows admitting cycles as the ones presented in [3]. *For loops*, being the simplest instance of a cycle, can be modelled with our current framework, although we would have to make sure that every step creates a new state, otherwise, even though the verification process would be done correctly, the reachability graph would be incomplete. *While loops* on the other hand are more complex to handle. Their specifications can also be done with our framework, but in this case nothing grants as termination. For this aim, we would need to make use of decidability classes. In [3] the authors present three of them, these results can potentially offer us the tools to use our approach to verify more complex classes of workflows. Nonetheless, modifications in the definition of algorithm 2 may also be required.

References

[1] Clara Bertolissi, Daniel Ricardo Dos Santos, and Silvio Ranise. 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*, pages 297–308, 2015.

[2] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Quantifier elimination for database driven verification. *arXiv preprint arXiv:1806.09686*, 2018.

[3] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Verification of data-aware processes via array-based systems (extended version). *arXiv preprint arXiv:1806.11459*, 2018.

[4] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Formal modeling and smt-based parameterized verification of data-aware bpmn. In *Business Process Management: 17th International Conference, BPM 2019, Vienna, Austria, September 1–6, 2019, Proceedings 17*, pages 157–175. Springer, 2019.

[5] Diego Calvanese, Silvio Ghilardi, Alessandro Gianola, Marco Montali, and Andrey Rivkin. Smt-based verification of data-aware processes: a model-theoretic approach. *Mathematical Structures in Computer Science*, 30(3):271–313, 2020.

[6] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards smt model checking of array-based systems. In *Automated Reasoning: 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings 4*, pages 67–82. Springer, 2008.

[7] Silvio Ghilardi and Silvio Ranise. Mcmt: A model checker modulo theories. In *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*, pages 22–29. Springer, 2010.

[8] Daniel Ricardo dos Santos and Silvio Ranise. A survey on workflow satisfiability, resiliency, and related problems. *arXiv preprint arXiv:1706.07205*, 2017.

[9] Qihua Wang and Ninghui Li. Satisfiability and resiliency in workflow systems. In *European Symposium on Research in Computer Security*, pages 90–105. Springer, 2007.