Authorization Enforcement in Workflows: Maintaining Realizability Via Automated Reasoning

Jason Crampton[1], Michael Huth[2] and Jim Huan-Pu Kuo[2]

[1] Information Security Group, Royal Holloway
University of London
jason.crampton@rhul.ac.uk
[2] Department of Computing
Imperial College London
m.huth,jimhkuo@imperial.ac.uk

Abstract

We investigate automated reasoning techniques as a means of supporting authorization enforcement functions of security-aware workflow management systems. The aim of such support is that one may statically or dynamically guarantee the realizability of a workflow instance given the security constraints of the underlying workflow specification.

We develop two such automated reasoning methods and experimentally evaluate their suitability for giving such support. One method uses a propositional encoding of realizability implemented through binary decision diagrams, another method uses a linear-time temporal logic encoding implemented via bounded model checking.

We chose these particular methods and implementations since they render representations that, at least in principle, capture many potential solutions so that dynamic guarantees of realizability can be made through efficient queries on these representations. Preliminary experimental results identify issues of scalability and of balancing flexibility in task allocation with complexity of computing such allocations.

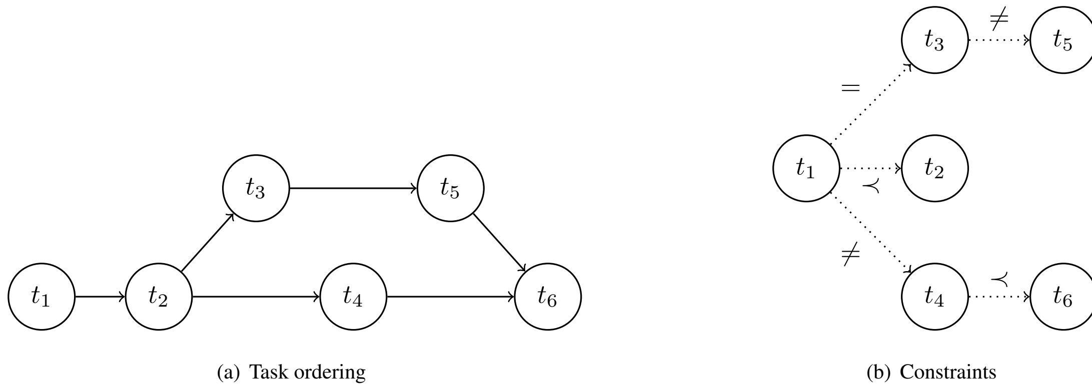
1 Introduction

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 workflow management systems or business process management systems.

A workflow typically specifies the tasks that comprise a business process and the order in which those tasks should be performed. Moreover, it is often the case that some form of access control should be applied to the execution of tasks. Hence, most workflow management systems may implement security controls that enforce authorization rules and business rules, in order to comply with statutory requirements or best practice. It is such "security-aware" workflows that will be the focus of this paper. Among the most useful security controls are:

- *user/task authorization* constraints, which specify which users may, in principle, execute what tasks;
- *binding of duty* (BoD) constraints, which require that certain tasks be executed by the same user in any given workflow instance;
- *separation of duty* (SoD) constraints, which require that certain tasks be executed by different users in any given instance of the workflow.

An illustrative example of a constrained workflow for purchase order processing is shown in Fig. 1. The purchase order is created and approved (and then dispatched to the supplier). The supplier will present an invoice, which is processed by the create payment task. When the supplier delivers the ordered goods, a goods received note must be signed and countersigned; only then may the payment be approved. A workflow specification need not be linear: the processing of the goods received note and of the invoice can occur in parallel, for example.



(a) Task ordering

(b) Constraints

t_1	create purchase order
t_2	approve purchase order
t_3	sign goods received note
t_4	create payment
t_5	countersign goods received note
t_6	approve payment
$\neq$	users performing the tasks must be different
$=$	users performing the tasks must be the same
$\prec$	user performing the second task must be senior to the user performing the first

(c) Figure legend

Figure 1: A simple constrained workflow for purchase order processing

In addition to constraining the order in which tasks are to be performed, some business rules are specified to prevent fraudulent use of this workflow. These rules take the form of constraints on users that can perform pairs of tasks in the workflow: for example, that the same user must not sign and countersign the goods received note.

The aggregate effect of such constraints may make it impossible to find an allocation of tasks to users and satisfy all the constraints. In other words, it may be that a workflow is rendered unrealizable by the inclusion of security controls. Hence, it is important to be able to determine whether a workflow specification can be realized.

There are different ways in which a workflow management system might choose to allocate tasks to users. These “execution models” give rise to different realizability problems but share the need to guarantee the continued realizability of a workflow instance. Hence, efficient decision procedures for workflow realizability are needed.

In this paper, we consider methods by which an authorization enforcement engine for workflow management systems might be designed. By construction, these methods should maintain the realizability of a workflow instance during its execution. We describe two such methods – a decision procedure and a search procedure – that can be called by such an engine. In particular, we explain how we can use

binary decision diagrams (BDDs) to build a decision procedure and bounded model checking to build a search procedure. We then describe our experimental work that compares the relative merits of these two methods.

Outline of paper. In Section 2, we present technical background of security-aware workflow systems. Two methods for supporting authorization enforcement functions for workflow instances, and their encodings through automated reasoning methods, feature in Section 3. Preliminary experimental data for implementations of these encodings are reported in Section 4. A brief discussion and our conclusions make up Section 5.

2 Preliminaries

We recall the definition of a constrained workflow authorization schema [2], which has formed the basis for a number of papers on workflow realizability, for example [1, 6].

Definition 1. *A* constrained workflow authorization schema $\mathcal{AS}$*, is a tuple* $(T, \leq, U, A, C)$ *where*

- T *is a set of tasks and* $(T, \leq)$ *is a partial order,*
- U *is a set of users and* $A \subseteq T \times U$ *an authorization relation,*
- C *is a finite set of* entailment constraints*, tuples of form* $(D, t \to t', \rho)$ *where* $D \subseteq U$*,* $t, t' \in T$ *and* $\rho \subseteq U \times U$*.*

The order $t \leq t'$ models that either t equals t' or task t has to be completed before task t' begins. Thus $\leq$ models *temporal* constraints on task execution. The authorization (t, u) in A models that user u is, at least in principle, authorized to execute task t. As we will see, the authorization enforcement engine may not allow an authorized user to execute a task because doing so would render the workflow instance unrealizable. An entailment constraint $(D, t \to t', \rho)$ models that if user u executes task t and u is from target set D, then the user u' who executes task t' (and who need not be from set D) must be related to u in the manner specified by ρ, i.e. (u, u') must be in ρ. For example, when D equals U the entailment constraint models BoD when ρ equals $=$, and it models SoD when ρ equals $\neq$. Note that entailment constraints $(D, t \to t', \rho)$, in and of themselves, do not impose any temporal order on the relative occurrence of t and t'.

2.1 Workflow Realizability

It is apparent that the existence of an authorization policy and entailment constraints may mean that there is no possible allocation of users to tasks. Hence, an important, practical question is whether a workflow authorization schema $\mathcal{AS}$ is *realizable* (also known as *satisfiable* in the literature). For our kind of schema, realizability means that one can allocate all tasks t in T to users in U such that all schema constraints (temporal order, authorization, and entailment) are satisfied. We now define this notion formally.

Definition 2. *Let* $\mathcal{AS}$ *denote a constrained authorized workflow schema as above. Then* $\mathcal{AS}$ *is* realizable *if there exists a total function* $\alpha\colon T \to U$ *such that*

- $(t, \alpha(t))$ *is in* A *for all t in* T*,*
- *for all* $(D, t \to t', \rho)$ *in* C*, if* $\alpha(t)$ *is in* D*, then* $(\alpha(t), \alpha(t'))$ *is in* ρ*.*

In other words, a workflow schema is realizable if there exists an allocation of users to tasks such that each user is authorized and all entailment constraints are satisfied. The reason that α suffices as a solution for realizability of $\mathcal{AS}$ is that our schema allows for the decoupling of temporal orderings from other constraints, and partial orders are always realizable ("linearizable"). We write $\mathsf{Sol}(\mathcal{AS})$ to denote the set of all functions α that realize the workflow $\mathcal{AS}$; this is the *solution space* of $\mathcal{AS}$, which may be empty.

2.2 Executing Workflows

A workflow management system (WfMS) is responsible for instantiating workflow schemas. The WfMS is also responsible for managing the execution of the tasks in a *workflow instance*. In particular, the WfMS will maintain a pool of *ready tasks*: the set of ready tasks in a workflow instance is the set of minimal tasks (with respect to the ordering on T) that have not yet been completed. Using the example in Fig. 1, the ready tasks once t_2 has been performed, for example, are t_3 and t_4; if t_4 is then performed, the set of ready tasks will be $\{t_3, t_6\}$.

In a workflow instance, the user/task allocation may be done in different ways [4].

- The WfMS creates a task list to which authorized users are allocated when a workflow is instantiated.
- The WfMS allocates authorized users to only those tasks that are presently ready.
- The WfMS maintains a pool of ready tasks from which users select tasks to execute.

We refer to these execution models as *static task allocation*, *dynamic task allocation* and *task selection*, respectively.

3 Two automated reasoning methods for authorization enforcement

The WfMS must incorporate a module, which we call the *authorization enforcement function* (AEF), that can ensure that

- a workflow instance is completed by users who are authorized for the respective tasks they perform,
- all constraints are satisfied, and
- the workflow instance completes.

The first of these responsibilities is a standard one for access-control functions and we will assume that it can be performed efficiently. The interesting question is how to implement the remaining functionality of the AEF.

The nature of the AEF will be determined by the execution model. In particular, there is an important distinction between static task allocation and the other two execution models. With static task allocation, the AEF computes a single mapping α of users to tasks, meaning that a single check for realizability is performed. Precomputing such a mapping maintains realizability by construction but does not allow for the modification of user-task bindings (which may perhaps be required for load-balancing, for example).

In contrast, no "up-front" computation of α is performed for dynamic task allocation and task selection. Instead, the AEF must perform a series of realizability checks on modified versions of the

workflow schema $\mathcal{AS}$. Once a task t has been performed by user u, then we transform the authorization relation of $\mathcal{AS}$ so that the only authorized user for t is u. We then determine the realizability of the modified schema. Henceforth, we only consider the task selection execution model, since the design of our AEF can be readily modified to accommodate the dynamic task allocation model.

The AEF traps all user requests to execute tasks, makes a decision on requests, and enforces that decision. We may model the AEF mathematically as a function of type

$$\mathsf{AEF}\colon \mathsf{accessRequest} \times \mathsf{state} \to \mathsf{decision} \times \mathsf{state} \tag{1}$$

where $\mathsf{accessRequest}$ is the set of request events the AEF has to process (here, access requests of form (t,u) in $T \times U$), state is an internal state that AEF maintains, and $\mathsf{decision}$ is the set of access-control decisions that AEF can make (here either $grant$ or $deny$). In other words, the AEF may inspect its internal state when making a decision on the current access request, and it may possibly alter its state as a result of that decision.

In the remainder of this paper we explore how such an AEF can be designed so that it may make access-control decisions that maintain the *realizability* of a constrained authorized workflow schema $\mathcal{AS}$. Concretely, we will discuss how automated reasoning tools may be used to give an AEF the ability to maintain realizability if at all possible. In particular, the state of an AEF will need to contain information that supports the maintenance of realizability of the workflow.

A key challenge in using automated reasoning tools is here that they should not incur a computational cost that would lead to unacceptable delays of access control decisions. It is this design constraint that will suggest to us methods that may precompute a representation of a large portion of the solution set, so that dynamic requests can be decided by an efficient inspection (and perhaps adjustment) of that representation. A formula of propositional logic, for example, may not be a suitable representation: although it can capture the entire solution space, querying it may involve a full SAT check that may simply take too long to complete in this application context.

3.1 Constructing an AEF with a Decision Procedure

We now describe how to construct an AEF from any decision procedure for workflow realizability so that this AEF maintains realizability whenever it grants access requests. Let $\mathcal{AS}$ denote a constrained authorized workflow schema as above. We assume the state σ maintained by the AEF to be a list of pairs of form $((u,t),d)$, where (u,t) is a request and d the decision the AEF made on request (u,t). In particular, there is at most one pair in σ with first component (u,t) – we assume that repeated tasks are distinct in $\mathcal{AS}$ – and we can extract from σ all requests to execute tasks that have been granted.

Let $\sigma_{\mathrm{complete}}$ be the set of tasks in T such that there exists an entry in σ of the form $((u,t), grant)$. For $t \in \sigma_{\mathrm{complete}}$, we write $\sigma(t)$ to denote the user that was granted permission to execute t. [1] We write $\sigma_{\mathrm{incomplete}}$ for $T \setminus \sigma_{\mathrm{complete}}$. We define

$$\mathcal{AS}[\sigma] \stackrel{\mathrm{def}}{=} (T, \leq, U, A[\sigma], C), \text{ where}$$
$$A[\sigma] \stackrel{\mathrm{def}}{=} (A \cap (\sigma_{\mathrm{incomplete}} \times U)) \cup \{(t, \sigma(t))\colon t \in \sigma_{\mathrm{complete}}\}\,.$$

In other words, for all t in $\sigma_{\mathrm{complete}}$, we replace all instances of (t,u) occurring in A with the sole entry $(t, \sigma(t))$, and leave all instances of t in $\sigma_{\mathrm{incomplete}}$ untouched in A.

Having established these concepts and notation, we can now sketch one possible approach to maintaining the realizability of $\mathcal{AS}$ through an AEF that is consistent with the type declared in (1). The

[1] Although $\sigma(t)$ might be a set of users, we assume that tasks are unique and so repeated tasks are differentiated through their instances.

```
(decision,state) AEF-DP(schema AS,state σ,accReq (t,u))
{
  if ((t,u) ∈ A && isRealizable(AS[σ | ((u,t),grant)]))
    { return (grant, σ | ((u,t),grant)); }
  else
    { return (deny, σ | ((u,t),deny)); }
}
```

Figure 2: An AEF incorporating a decision procedure for workflow realizability

pseudocode for `AEF-DP` is depicted in Fig. 2. The decision procedure `isRealizable` takes a workflow schema $\mathcal{AS}$ as input and returns `true` if and only if $\mathsf{Sol}(\mathcal{AS})$ is non-empty (i.e. returns `true` if and only if $\mathcal{AS}$ is realizable).

We now describe the behavior of `AEF-DP`, where we write $\sigma \mid x$ for the state that appends to list σ the item x of appropriate type. A request (t,u) is denied if either (t,u) is not in the authorization relation A of $\mathcal{AS}$,[2] or if `isRealizable`, when supplied with input $\mathcal{AS}[\sigma \mid ((u,t,),\mathit{grant})]$, returns false – meaning that there is no function α in $\mathsf{Sol}(\mathcal{AS})$ that maps u to t and $\sigma(t')$ to t' for all t' that have been executed – as granting it would make the remaining workflow unrealizable. Otherwise, the request is granted. In any event, σ is updated to reflect the decision made.

The crucial invariant that this AEF guarantees is that

> "**Invariant:** All grants of access requests mean that the workflow $\mathcal{AS}$ is realizable in the updated state."

One possible drawback of this approach is that the decision procedure `isRealizable` needs to be called each time an access-control request is made. As already discussed, one limiting factor will certainly be the space and time requirements for such a decision procedure. Therefore, we will now explore whether automated reasoning tools can be devised that fare better in this regard.

3.2 Constructing an AEF with Solution Sets

The method `AEF-DP` maintains the realizability of a workflow, but makes no use of "witness" information for such realizability. One price we pay for this is that we need to recompute realizability decisions each time a request is processed by `AEF-DP`.

Hence, we now discuss an alternative approach that uses a *search procedure* to compute a (representation of a) subset of $\mathsf{Sol}(\mathcal{AS})$. The procedure relies on an abstraction of $\mathsf{Sol}(\mathcal{AS})$. More precisely, it computes two partitions

$$T = \bigcup_{i=0}^{n} T_i \qquad \text{and} \qquad U \supseteq \tilde{U} = \bigcup_{i=0}^{n} U_i \tag{2}$$

where *all functions* $\alpha\colon T \to U$ such that $(t_i,\alpha(t_i))$ belongs to $T_i \times U_i$ for all $0 \leq i \leq n$ belong to $\mathsf{Sol}(\mathcal{AS})$. The intuition here is that we may assign to any task in T_i any user in U_i, and that we can be sure that this will not interfere with any constraints within (T_i,U_i) nor across any of the set-valued task/user pairs (T_j,U_j). Note that (2) partitions task set T but only partitions a subset $\tilde{U}$ of users of U that will be allocated to tasks in that workflow instance. In effect, this is an under-approximation of

[2] In the interests of brevity, our pseudocode does not include sanity checks such as ensuring that the requested task has not already been performed. As already mentioned, we also assume that multiple occurrences of the same task are distinguishable in the schema.

```
(decision,state) AEF-SS(schema 𝒜𝒮,state (Σ,σ),accReq (t,u))
{
  if (there exists (T,U) ∈ Σ such that t ∈ T and u ∈ U)
    { return (grant,(Σ,  σ | ((u,t),grant))) }
  else
    { return (deny,(Σ,  σ | ((u,t),deny))) }
}
```

Figure 3: Constructing an AEF using a static prepartitioning of solutions

$\mathsf{Sol}(\mathcal{AS})$ as it represents a subset of that space of solutions and does not represent functions that aren't solutions.

Given a method `getAbs` for computing such partitions, we can write a second AEF, `AEF-SS`, pseudocode for which is shown in Fig. 3. This approach assumes the state is an ordered pair (Σ,σ), where Σ is some representation of two partitions as in (2) computed using `getAbs`, and (as before) σ is a list that records which requests have been processed with what decisions. In particular, σ records which tasks have already been allocated to which users by `AEF-SS`.

Given a request (u,t), `AEF-SS` inspects whether u and t belong to the same task/user pair computed by `getAbs`, i.e. whether there is some i so that u is in U_i and t in T_i. If so, access is granted; otherwise it is denied. In particular, the Σ part of the state never changes and `AEF-SS` never has to recompute realizability information. However, it is possible that `AEF-SS` may deny a request that would not prevent the completion of a workflow instance.

4 Implementation and Evaluation

In this section, we describe how the procedures `isRealizable` and `getAbs` can be constructed. For the `isRealizable` procedure, we encode the realizability problem as an instance of SAT for propositional logic (PL); we do this since we want to test whether BDDs might serve as an effective representation of the solution space. For the procedure `getAbs` we capture this also as a SAT instance but in linear-time temporal logic (LTL), as done in [3]. We use LTL and a bounded model checker here as we can instrument the LTL encoding so that it precomputes partitions as in (2) that can be used as a basis for `AEF-SS`. We also report on experimental work that tests the performance of our methods and these encodings when applied to synthetic (randomly generated) workflow schemas.

4.1 Procedure `isRealizable`

Formula $\eta_{\mathcal{AS}}$ encodes the realizability problem for $\mathcal{AS}$ as an instance of SAT for PL, where models of $\eta_{\mathcal{AS}}$ correspond to elements of $\mathsf{Sol}(\mathcal{AS})$ and vice versa. This encoding is shown in Fig. 4. Its set of propositional variables is

$$\{x_{(t,u)} \mid (t,u) \in A\}$$

where we define the sets of users U_t and $u.\rho$ as

$$U_t = \{u \in U \mid (t,u) \in A\} \quad (3)$$
$$u.\rho = \{u' \in U \mid (u,u') \in \rho\} \quad (4)$$

This encoding is sound and complete since we can show that $\eta_{\mathcal{AS}}$ is satisfiable if and only if $\mathsf{Sol}(\mathcal{AS})$ is non-empty, i.e. $\mathcal{AS}$ is realizable. The intuition behind the encoding is that t *may be allocated* to u if $x_{(t,u)}$ is true, and that t *must not be allocated* to u if $x_{(t,u)}$ is false.

$$\begin{aligned}
\eta_{bind} &= \bigwedge_{t\in T}\bigvee_{u\in U_t} x_{(t,u)} \\
\eta_C &= \bigwedge_{(D,t\to t',\rho)\in C} \eta_{(D,t\to t',\rho)} \\
\eta_{(D,t\to t',\rho)} &= \bigwedge_{u\in D\cap U_t}\left(x_{(t,u)} \to \bigwedge_{u'\in U_{t'}\setminus u.\rho} \neg x_{(t',u')}\right)
\end{aligned} \tag{5}$$

Figure 4: Encoding $\eta_{\mathcal{AS}} \stackrel{\text{def}}{=} \eta_{bind} \wedge \eta_C$: workflow realizability as instance of SAT for PL

Specifically, formula η_{bind} specifies that all tasks may be allocated to some user – a necessary requirement for realizability. Formula η_C simply stipulates that all formulas $\eta_{(D,t\to t',\rho)}$ that encode entailment constraints must be true. And such a formula $\eta_{(D,t\to t',\rho)}$ states that if a user u from set D may be allocated to task t, then all users u' that are authorized to execute task t' but are not in a relationship to u via ρ are such that they must not be allocated to t'. Note that "It is not the case that u' may allocate task t' " is equivalent to "It is the case that u' must not be allocated to task t' ". The need for this indirection is that the variables do not represent the modality "must be allocated".

A Boolean function (and so $\eta_{\mathcal{AS}}$ as well) can be represented as a binary decision diagram (BDD), a DAG-type data structure that eliminates redundancies in binary decision trees; and this representation is unique for a fixed order of variables in the BDD. The main reason why we are interested in BDDs here is that one can efficiently compute specializations of BDDs (in which the truth values of some variables are fixed) in order to decide the realizability of a workflow in an updated state. Thus we could implement non-initial calls to `isRealizable` efficiently relative to the complexity of computing the "initial" BDD from $\eta_{\mathcal{AS}}$. Our experiments therefore focus on the latter computation.

Given $\eta_{\mathcal{AS}}$, we first synthesize from it a BDD $B_{\mathcal{AS}}$ (using a standard BDD library `JavaBDD`, which relies on the CUDD implementation in `C`, and its default variable ordering) and then check (in constant time) whether that BDD is equal to the canonical BDD that contains only leaf 0 (and so represents "unsatisfiable"). If and when this BDD has been built, we can implement the call to `isRealizable` in Figure 2 by simply computing the specialization of this BDD that eliminates one variable.

4.2 Procedure `getAbs`

Our implementation of `getAbs` is through a reduction of realizability of $\mathcal{AS}$ to SAT for the NP-complete fragment [5] $\mathsf{LTL}(\mathsf{F})$ of LTL. We quickly review the syntax and semantics of $\mathsf{LTL}(\mathsf{F})$: Given a finite set AP of atomic propositions (this is $T \cup U$ here), the propositional temporal logic $\mathsf{LTL}(\mathsf{F})$ is generated by the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{F}\,\phi$$

where p is from AP and F is the temporal connective "Future" such that $\mathsf{F}\,p$ states that p will be true at some point in the future.

A *model* of a formula ϕ is an infinite sequence of states $\pi = s_0 s_1 \dots$, where each s_i is a subset of AP. We write $\pi \models \phi$ if π is a model for ϕ. We say that a formula ϕ is *satisfiable* if and only if it has a model. We write π^i to denote the infinite suffix $s_i s_{i+1} \dots$ of π. The formal semantics of formulas is then given in Figure 5.

$\pi \models p \quad \text{iff} \quad p \in s_0$ $\qquad$ $\pi \models \neg\phi \quad \text{iff} \quad \text{not } \pi \models \phi$
$\pi \models \phi_1 \wedge \phi_2 \quad \text{iff} \quad (\pi \models \phi_1 \text{ and } \pi \models \phi_2)$ $\qquad$ $\pi \models \mathsf{F}\,\phi \quad \text{iff} \quad \text{there is } i \geq 0\colon \pi^i \models \phi$

Figure 5: Formal semantics of temporal logic $\mathsf{LTL}(\mathsf{F})$ over models $\pi = s_0 s_1 \ldots$

$$\begin{aligned}
\delta_{\mathsf{F}T} &\stackrel{\text{def}}{=} \bigwedge_{t\in T} \mathsf{F}\,t \\
\delta_{\mathsf{G}U} &\stackrel{\text{def}}{=} \mathsf{G}(\bigvee_{u\in U} u) \\
\delta_{A} &\stackrel{\text{def}}{=} \bigwedge_{t\in T} \mathsf{G}\Big(t \to \neg(\bigvee_{(t,u)\notin A} u)\Big) \\
\delta_{C} &\stackrel{\text{def}}{=} \bigwedge_{(D,t\to t',\rho)\in C} \delta_{(D,t\to t',\rho)} \\
\delta_{(D,t\to t',\rho)} &\stackrel{\text{def}}{=} \bigwedge_{u\in D} \big(\mathsf{F}\,(t\wedge u)\big) \to \mathsf{G}\Big(t' \to \neg(\bigvee_{(u,u')\notin\rho} u')\Big) \\
\delta_{\mathsf{F}U} &\stackrel{\text{def}}{=} \bigwedge_{u\in U} \mathsf{F}\,u
\end{aligned}$$

Figure 6: Encoding $\delta_{\mathcal{AS}} \stackrel{\text{def}}{=} \delta_{\mathsf{F}T} \wedge \delta_{\mathsf{G}U} \wedge \delta_A \wedge \delta_C \wedge \delta_{\mathsf{F}U}$ of [3]: workflow realizability in $\mathsf{LTL}(\mathsf{F})$

We use the usual abbreviations for disjunction ($\vee$), implication ($\to$), logical equivalence ($\leftrightarrow$) and the "Global" temporal connective $\mathsf{G}\,\phi$, which stands for $\neg\mathsf{F}\,\neg\phi$ (the informal interpretation being "always ϕ").

The realizability of $\mathcal{AS}$ we encode as a SAT instance for $\mathsf{LTL}(\mathsf{F})$ formula $\delta_{\mathcal{AS}}$ shown in Fig. 6. Its satisfiability is decided using the model checker NuSMV on a fully connected model, formula $\neg\delta_{\mathcal{AS}}$, and in an incremental bounded model-checking mode. The set of variables for this encoding is the disjoint union $T \cup U$. The interpretation of a variable t (respectively, u) being true in state s_i is that it is in set T_i (respectively, U_i) of the constructed partition. Thus the $\delta_{\mathcal{AS}}$ encoding allows the possibility that several tasks and users may hold in a state.[3]

If the model checker returns a "counterexample", a finite trace of states $s_0 s_1 \ldots s_n$ that represents a "lasso" path π that makes $\delta_{\mathcal{AS}}$ true, then we can derive a partition

$$T_i = s_i \cap T \qquad\qquad U_i = s_i \cap U \tag{6}$$

and show (see [3]) that all α that allocate tasks consistent with all (T_i, U_i) pairings are in $\mathsf{Sol}(\mathcal{AS})$.

We now discuss this encoding in greater detail. Formula $\delta_{\mathsf{F}T}$ demands that all tasks have to be true at some state, whereas $\delta_{\mathsf{G}U}$ ensures that all states make some user(s) true. Formula δ_A indirectly captures the authorization relation A: for all tasks t, if t is true at some state then no users that are un-authorized to execute t can be true at that state. The reason for this indirect encoding is that we need to rule out that user and task groupings at a state violate any constraints, and that we cannot control or predict these groupings.

[3] An encoding of workflow realizability in $\mathsf{LTL}(\mathsf{F})$ in which we insist that a single user and task are executed in all states has poor model checking results [3].

Formula δ_C states that all entailment contraints have to be met. And formula $\delta_{(D,t\to t',\rho)}$ captures such an entailment constraint. If a user u from set U is grouped with task t at some state, then at all states that make t' true there are no users u' true there which are not in relationship ρ with u. Again, this indirection is needed in order for the model checker to discover such groupings of users and tasks at states.

Intuitively, it is desirable to have states s_i in which there are as many tasks and users as possible, as this gives us more flexibility when dealing with access requests. Similarly, we want this search procedure to have the tendency of accommodating, and so possibly allocating, as many users and tasks in the sets T_i and U_i. This tendency is actively encouraged through the conjunct $\delta_{\mathsf{F}U}$ in encoding $\delta_{\mathcal{AS}}$. The intuition behind the inclusion of this conjunct is that we use a bounded model checker that will find the shortest possible "lasso" trace that represents a model of the formula. So the model checker will indeed try to pack as many users into states as possible to capture a solution, and will put all those users that were not needed for the solution into a "junk" state, and only into one such junk state. We found this to be beneficial when compared to an encoding that does not include this conjunct [3].

4.3 Experimental data

We now discuss our experimental results, which compare the performance of the BDD approach ($\eta_{\mathcal{AS}}$ for `isRealizable` in `AEF-DP`) to a bounded model-checking approach ($\delta_{\mathcal{AS}}$ for `getAbs` in `AEF-SS`) on randomly generated workflows $\mathcal{AS}$ (be they realizable or not). These experiments were conducted on the same Ubuntu Linux machine with Intel® Core™ 2 Duo Processor at 2.8 Gigahertz and 4 Gigabytes of RAM.

We now describe the set of configurations for the workflow schemas $\mathcal{AS}$ used in our experiments. Each $\mathcal{AS}$ was generated according to three parameters: the number of users, the *authorization density*, and the *constraint density*. In each configuration the number of tasks was equal to the number of users, taking values $10, 20, \ldots, 140, 150$. Authorization density is defined to be the ratio of $|A|$ to the product of $|U|$ and $|T|$, where the latter represents the maximum possible cardinality of A. The authorization densities we considered in our experiments are 0.1, 0.5, and 1.0, therefore ranging from a rather sparse authorization policy through to one in which all users are authorized to perform all tasks. The constraint density is defined to be the ratio of $|C|$ to $|U|$. We let this value range over 0.05, 0.10, and 0.20. The reason for choosing these lower values, but still having a good spread within that low range, is that higher values of the constraint density tend to produce only unrealizable *ramdomly* generated $\mathcal{AS}$ and we are interested in realizable $\mathcal{AS}$ as we mean to support such realizability as an invariant in an AES.

We present our results in graphical form in Figures 7 to 9. Each figure shows results for a different authorization density. In each figure, the y-axis represents the time (on a *logarithmic* scale) taken to determine realizability, where that time is the average time over 10 schemas $\mathcal{AS}$ for the respective configuration type. The x-axis represents the configuration type for our experiments. A configuration type has form `uu-cd` where `uu` is the number of users (and so the number of tasks as well) and `cd` is the constraint density. The absence of a bar for a given configuration type indicates that the experiment timed out after 20 minutes or ran out of memory.

Figure 9 suggests that the LTL approach outperforms the BDD approach for high values of `ad` such as 1.0: the latter cannot even generate BDDs for workflows with more than 20 users whereas the LTL approach can do this for at least 150 users. Looking at the data on Figures 8 and 7, we can see that the BDD approach seems to catch up to the LTL approach as the value of `ad` becomes lower. The effect of `ad` seems to be reversed in both approaches.

We now analyze how both approaches vary with the value of `cd`. Inspecting the three figures, we note that in each figure its three "zones" of constraint densities have a very similar shape for both approaches. Therefore, we can hypothesize that, in general, this value has less of an effect and the same

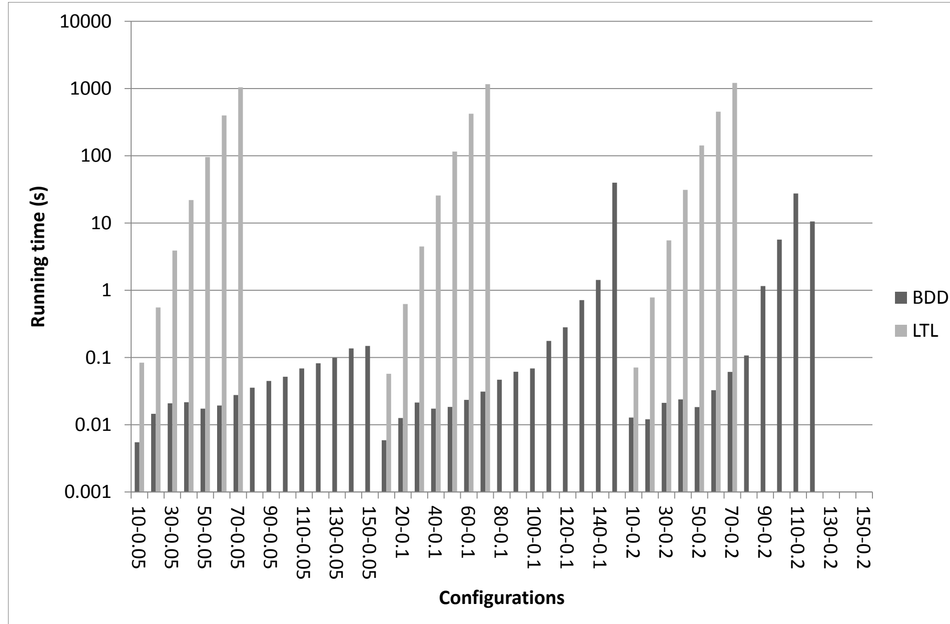


Figure 7: Comparison of time taken to determine realizability using BDDs vs. LTL model checking for authorization density 0.1

type of effect on both the BDD and LTL approaches.

Finally, both approaches find it difficult to scale the decidability of realizability in that the running time appears to grow exponentially in the number of users (and tasks), as the effort resembles a linear function on a logarithmic scale. For the LTL approach, we tried to determine its limits when `ad` equals 0.5 and `cd` equals 0.1. These experiments (not reported here) suggest that this approach fails consistently on our machine for models with more than 230 users.

5 Conclusions

We presented constrained authorized workflow schemas and motivated the need for workflow management systems to maintain the realizability of such “security-aware” workflows. We suggested two authorization enforcement functions that use automated reasoning methods in order to maintain realizability as an invariant of task execution.

One of these methods relies on a decision procedure for realizability encoded in propositional logic. As a workflow instance executes, this costly procedure needs to be called at each access request instance. Unfortunately, our attempt to circumvent this need through the synthesis of BDDs and their dynamic specialization leads to discouraging experimental results for the build of the initial BDD.

The second method is already reported in [3] and, in effect, computes a subset of the set of all re-

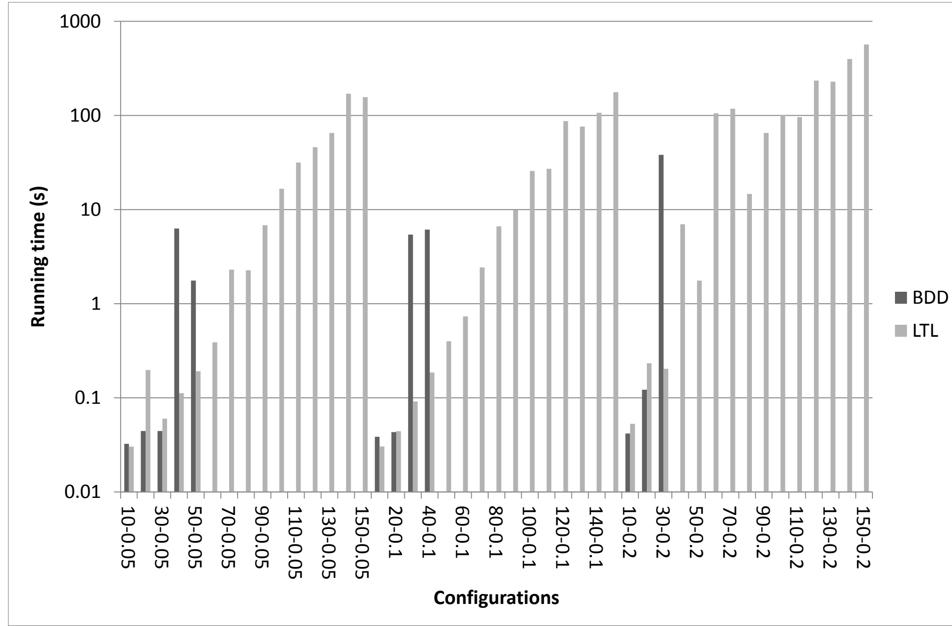


Figure 8: Comparison of time taken to determine realizability using BDDs vs. LTL model checking for authorization density 0.5

alizability solutions for a workflow instance, where this subset is defined by a sequence of task-user subsets. Such a sequence can be computed using an appropriately configured bounded model-checker for linear-time temporal logic with a suitable encoding derived from the workflow schema. The experimental results for this approach are more encouraging but at least two issues need to be resolved in order to make this approach viable in practice. Firstly, we need to develop refinements of this approach in order to make the model checking more scalable, for example through the use of further abstraction techniques.

Secondly, we need to investigate whether the compact "Boolean" subset of solutions computed by the model checker can, implicitly represent even more solutions and so make the authorization enforcement function more flexible. Delegation models of workflow schemas [4], where users may delegate task execution rights to other users, are just one motivation for such increased flexibility. This second issue has also a more general form: we want to understand the trade-offs between the complexity of computing realizability information that supports an authorization enforcement function and the frequency of denying access requests that, if granted, could in principle lead to realizable workflow instances.

Of course, there are many other approaches to automated reasoning that we may test for their suitability of supporting workflow realizability. Perhaps an incremental SAT solver may allow for a relatively quick decision of the realizability of access requests; we mean to investigate this in future experimental work.

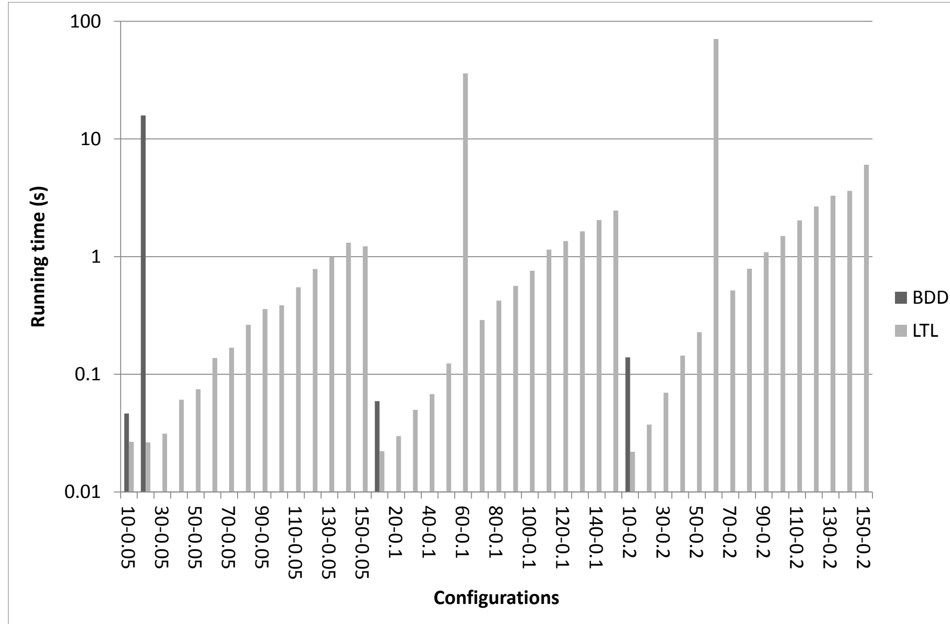


Figure 9: Comparison of time taken to determine realizability using BDDs vs. LTL model checking for authorization density 1.0

The authorized workflow schemas studied in this paper share with existing approaches in the literature that the population of users is already part of the schema. It seems undesirable, somehow, to do the automated reasoning over such a concrete population. In future work, we therefore mean to investigate whether such automated reasoning can be done over a dynamically expanding, symbolic set of users. The aim would be to compute a user/task assignment for symbolic users, which then leaves us with the orthogonal problem of mapping symbolic users into concrete user populations, be it statically or at runtime. Our preliminary study of this new approach suggests that one may fruitfully use constraint satisfaction solvers or efficient instances of colorability problems for the computation of such symbolic solutions.

References

[1] David Basin, Samuel J. Burri, and Guenter Karjoth. Obstruction-free authorization enforcement: Aligning security and business objectives. In *24th Computer Security Foundations Symposium (CSF 2011)*, pages 99–113, Cernay-la-Ville, France, 06 2011. IEEE.

[2] J. Crampton. A reference monitor for workflow systems with constrained task execution. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, pages 38–47, 2005.

[3] J. Crampton, M. Huth, and J. H.-P. Kuo. Authorized workflow schemas: Deciding realizability through $\mathsf{LTL}(\mathsf{F})$ model checking. Technical Report 3, Imperial College London, Department of Computing, May 2012. Submitted.

[4] J. Crampton and H. Khambhammettu. Delegation and satisfiability in workflow systems. In I. Ray and N. Li, editors, *Proceedings of 13th ACM Symposium on Access Control Models and Technologies*, pages 31–40, 2008.

[5] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32:733–749, July 1985.

[6] Qihua Wang and Ninghui Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.*, 13(4):40, 2010.