Modelling User Availability in Workflow Resiliency Analysis

John C. Mace
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
john.mace@ncl.ac.uk

Charles Morisset
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
charles.morisset@ncl.ac.uk

Aad van Moorsel
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
aad.vanmoorsel@ncl.ac.uk

ABSTRACT

Workflows capture complex operational processes and include security constraints limiting which users can perform which tasks. An improper security policy may prevent certain tasks being assigned and may force a policy violation. Deciding whether a valid user-task assignment exists for a given policy is known to be extremely complex, especially when considering user unavailability (known as the resiliency problem). Therefore tools are required that allow automatic evaluation of workflow resiliency. Modelling well defined workflows is fairly straightforward, however user availability can be modelled in multiple ways for the same workflow. Correct choice of model is a complex yet necessary concern as it has a major impact on the calculated resiliency. We describe a number of user availability models and their encoding in the model checker PRISM, used to evaluate resiliency. We also show how model choice can affect resiliency computation in terms of its value, memory and CPU time.

Categories and Subject Descriptors

H.4.1 [**Information Systems Applications**]: Office Automation—*workflow management*; I.6.5 [**Simulation and Modelling**]: Model Development—*modelling methodologies*

General Terms

Security, Reliability, Human Factors

Keywords

Workflow Satisfiability Problem, Markov Decision Process, Probabilistic Model Checker

1. INTRODUCTION

Workflows capture the logical processes of achieving business goals and are used in almost all business domains including finance, healthcare and eScience [5, 9, 12]. Although the definition of what a workflow is may vary within these domains, a workflow typically consists of atomic tasks (the work) that are logically ordered (the flow) to produce the required outcome [1].

Tasks must be assigned to users in order to be performed, often in-line with business rules and regulatory requirements defined within a workflow security policy [3]. A policy can limit task execution only to qualified users and limit access to (often) sensitive information which is needed by a user during the course of their duties. Workflow policies typically come with three main constraints *1*) user-task permissions stating which users can perform which tasks; *2*) separation of duties stating which tasks must be performed by distinct users to limit fraud and error [7]; *3*) binding of duties stating tasks that must be performed by the same user providing consistency and limiting the dissemination of information across multiple tasks [14].

A workflow security policy designer is tasked with defining a policy that not only meets all business security requirements but also allows the workflow to proceed to completion without hindrance [4, 15]. In practice this exercise is often not trivial in light of limited user numbers, large workflows and complex security requirements. Finding an assignment of users to tasks that allows a workflow to finish without being forced to terminate early or violate its security policy deems the workflow as *satisfiable*. It is the job of the policy designer to ensure such an assignment exists as an outcome of their drafted policy. Solving this problem, known as the *workflow satisfiablility problem* (WSP) is an ongoing research problem and is well studied in the literature [8, 20]. The WSP has been shown to be NP-hard meaning every combination of users to tasks may have to be tried before finding an assignment which satisfies a workflow.

The WSP is all well and good if one assumes users will always be available. If this *is* the case, a satisfiable workflow will always complete in every instance. However in practice, users become unavailable due to a number of reasons some of which may be unforeseen, e.g. preoccupation or sickness. It follows that a *satisfiable* workflow before execution may become *unsatisfiable* during its operation. The *workflow resiliency problem* extends the WSP by finding an assignment if one exists, to satisfy a given workflow under an assumption that some users may become unavailable.

Availability is usually defined in terms of *up-time*. That is, states of a system are labelled either *up* or *down*, and the fraction of time in the states labelled *up* corresponds to the system's availability. In [19], Meyer and Sanders provide a general approach to the definition of metrics such as this. In this paper we discuss a more specific notion of availability,

HotSoS '15, April 21 - 22, 2015, Urbana, IL, USA
Copyright 2015 ACM 978-1-4503-3376-4/15/04 ...$15.00
http://dx.doi.org/10.1145/2746194.2746201.

namely the availability of users, which can be present or not.

User unavailability in workflows was introduced by Wang and Li [20], who considered a qualitative approach where users are either available or not. A workflow is classified as k resilient if the workflow can still be satisfied regardless of which k users become absent. Mace et al [17] considered a more quantitative approach by assigning probabilities to users becoming absent. Calculating the resiliency of a workflow was shown to be equivalent to finding the optimal policy of a Markov Decision Process (MDP) [6]. This approach is practical in the sense it provides a level of resiliency (success rate) to workflows that must work but cannot be made fully resilient in every case.

It is useful then to provide tools and methodologies to workflow security policy designers enabling them to automatically calculate the resiliency offered by a workflow security policy. Such tools can be beneficial in a number of ways, they can *1*) enable policy comparability in terms of workflow resiliency; *2*) identify resiliency changes following policy modifications; *3*) ensure a policy provides the minimum requirement of resiliency; *4*) highlight extra user level requirements, and unworkable processes and policies; *5*) provide resiliency indicators and assurance to business leaders and process designers;

A well defined workflow consisting of ordered tasks and users constrained by a security policy is fairly straightforward to model [8, 17]. On the other hand, capturing user availability in a model relies on, and is only as good as, the availability inputs provided by the environment. This may for example involve users with non-deterministic or probabilistic availabilities for workflow tasks. Depending on the information available and other contextual inputs, a user availability model could be defined in multiple ways for the same workflow. Making the correct choice of model becomes a complex yet necessary issue as it has a major impact on the resiliency calculated for a workflow. A poor choice may result in a flawed and misleading resiliency value.

In this paper we model workflows and user availability using the model checking tool PRISM [16] which is able to calculate the resiliency of workflows modelled as MDPs. We do not state which choice of availability model to take but instead explore PRISM's capability to encode a variety of user models containing non-deterministic and probabilistic availabilities. We also show how model choice can impact resiliency computation, not only in terms of the resiliency value but also in terms of memory and CPU time.

The rest of the paper is as follows. Section 2 gives an overview of related work, Section 3 discusses the workflow satisfiability problem whilst Section 4 describes workflow resiliency. Section 5 discusses encoding workflows and user availability in PRISM, Section 6 provides analysis of how the choice between user availability models affects resiliency computation, whilst concluding remarks are given in Section 7.

2. RELATED WORK

Model checking has been used by Armando et al. [2] to formally model and automatically analyse security constrained business processes to ensure they meet given security properties. He et al. in [10] also use modelling techniques to analyse security constraint impact in terms of computational time and resources on workflow execution.

Herbert et al. in [11] model workflows expressed in BPMN as MDPs. The probabilistic model checker PRISM [16], is utilised to check various probabilistic properties such as reaching particular states of interest, or the occurrence and ordering of certain events. Assuming resiliency is the property we want to verify, a similar approach can be taken to calculate workflow resiliency. Quantitative access control using partially-observable MDPs is presented by Martinelli et al. in [18] which under uncertainty, aims to optimise the decision process for a sequence of access requests.

However, to the best of our knowledge, there is no current literature on the impact user availability model choice can have on the automatic analysis of workflow resiliency, which is the focus of this paper.

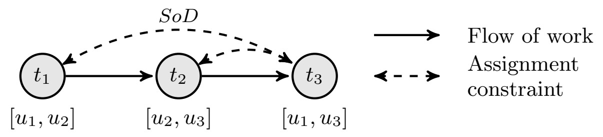

Figure 1: Running example workflow

3. WORKFLOW SATISFIABILITY

In this section we give a simple workflow definition before describing the process of assigning users to tasks whilst meeting security constraints, known as the workflow satisfiability problem (WSP).

3.1 Workflow

In the literature a workflow commonly consists of a partially ordered set of tasks $(T, <)$, such that for any two tasks $t, t' \in T$, if $t < t'$ then t must be performed before t' in any instance of the workflow [8, 20].

Running example. *As a running example to illustrate the different concepts presented here, we consider a simple workflow shown in Figure 1 where* $T = \{t_1, t_2, t_3\}$ *and* $t_1 < t_2 < t_3$, *meaning the only valid execution sequence is* $\langle t_1, t_2, t_3 \rangle$.

Next we define a set of users U that comes with a security policy over the set of tasks T. In general, a security policy is a triple $p = (P, S, B)$ where:

- $P \subseteq U \times T$ are *user-task permissions*, such that $(u, t) \in P$ if, and only if u is allowed to perform t.
- $S \subseteq T \times T$ are *separations of duty*, such that $(t, t') \in S$ if, and only if the users assigned to t and t' are distinct.
- $B \subseteq T \times T$ are *bindings of duty*, such that $(t, t') \in B$ if, and only if the same user is assigned to t and t'.

Running example. *We now consider a set of users* $U = \{u_1, u_2, u_3\}$ *and a security policy* $p_1 = (P_1, S_1, B_1)$ *that states:*

- $P_1 = \{(u_1, t_1), (u_2, t_1), (u_2, t_2), (u_3, t_2), (u_1, t_3), (u_3, t_3)\}$
- $S_1 = \{(t_1, t_3), (t_2, t_3)\}$
- $B_1 = \emptyset$

Figure 1 illustrates p_1, *where the dotted arrows signify the constraints given in* S_1. *A label* $[u_m, ..., u_n]$ *states the users that are authorised by* P_1 *to execute* t_i.

DEFINITION 1 (WORKFLOW). *A workflow is a tuple* $w = ((T, <), U, p)$, *where* T *is a partially ordered set of tasks,* U *is a set of users, and* p *is a security policy.*

3.2 Assignment

A *workflow assignment* is a relation $A \subseteq U \times T$, such that $(u_i, t_i) \in A$ is a *user-task assignment* indicating u_i is assigned to t_i. Informally, any A is *valid* if *a*) task ordering is

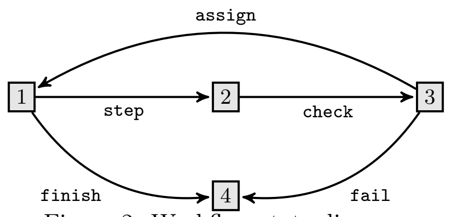

Figure 2: Workflow state diagram

respected; *b*) any user-task assignment is permitted; *c*) separation and binding constraints are respected; *d*) no task is executed twice.

A given assignment A is *complete* if it includes a user-task assignment for every task in the workflow, or *partial* otherwise. The WSP therefore consists of finding an assignment A that is both valid and complete which we denote as $A \vdash WSP$. In general the WSP has been shown to be NP-hard, in other words, all possible user-task assignments may have to be checked to solve it [20].

Imagine we want to find $A \vdash WSP$ for our running example and begin by assigning u_1 to t_1 and u_3 to t_2 to form the partial assignment $A' = \{(u_1, t_1), (u_3, t_2)\}$. Although this assignment is valid, there is no $u \in U$ such that $A' \cup \{(u, t_3)\} \vdash WSP$, meaning the workflow cannot finish correctly. However, with the partial assignment $\{(u_1, t_1), (u_2, t_2)\}$, we can add (u_3, t_3) to form a complete and valid assignment.

3.3 Quantitative Satisfaction

In [17], Mace et al took a quantitative approach by showing the solution to the WSP is equivalent to finding the optimal policy of a Markov Decision Process (MDP) [6]. The MDP presented is based on the state transition diagram shown in Figure 2, which depicts an abstracted view of a workflow assignment process. Each node represents one of four system states read as follows: *1*) ready to select user-task assignment; *2*) ready to check user-task assignment satisfies policy; *3*) policy checked; *4*) workflow terminated.

Each directed arrow between two states s and s' indicates a transition from s to s'. The five transitions are read as:

- `step`: get the next workflow task t and a perspective user u to assign to t
- `check`: check assignment (u, t) satisfies security policy
- `assign`: (u, t) satisfies security policy
- `fail`: (u, t) does not satisfy security policy
- `finish`: no more tasks to assign

Solving the MDP involves calculating the optimal value function on every possible state. Roughly speaking, the MDP takes the transition `step` for each $u \in U$ and next task t, assigning the user to t who maximises the expected reward collected by the process. The value function returns a reward of 1 if there exists $A \vdash WSP$, or 0 otherwise.

4. WORKFLOW RESILIENCY

Solving the WSP assumes users will always be available for future tasks, however in practice, sickness, vacation, heavy workloads, etc., can cause users to *fail.* It is important to take this into account when finding $A \vdash WSP$ for a given workflow. This is called the resiliency problem, whether a workflow can be satisfied even when some users become absent.

It may be the case that selecting a user-task assignment leads to a situation where all valid users for a future task will have failed or have a low probability of availability. To give a flavour for this, in our running example we choose $A = \{(u_1, t_1), (u_2, t_2), (u_3, t_3)\} \vdash WSP$. However, imagine u_3 has a very high probability of failing at or before t_3, observed from previous workflow logs. If u_3 does fail, t_3 cannot be reassigned to any other user meaning the workflow cannot finish. If we chose a different assignment $A' = \{(u_2, t_1), (u_2, t_2), (u_3, t_3)\} \vdash WSP$, intuitively the workflow is more resilient as t_3 can be reassigned to u_1 and still finish if u_3 did indeed fail.

4.1 Quantitative Resiliency

Wang and Li defined an approach to calculate a valid assignment if one exists, that is resilient to up to k users failing, in other words declaring a workflow to be either k resilient or not [20]. In many cases, finding an assignment for a workflow that is resilient to every combination of k user failures may be impossible, yet the workflow must still work. Yet finding a valid assignment that is resilient in 9 out of 10 cases is better than choosing a valid assignment that is resilient in only 1 out of 10 cases. In [17], Mace et al extend their approach by introducing probabilistic user failures and show that computing the optimal policy of an MDP is equivalent to finding $A \vdash WSP$ that maximises the value function. The value function returns $0 > v \leq 1$ if there exists $A \vdash WSP$ where v indicates the probability of the workflow to finish, or 0 otherwise.

4.2 User Availability

Understanding when users will and will not be available is an obvious requirement when calculating resiliency, which may be deduced from a mixture of operational logs, behavioural analysis and user submissions (known and tentative absences). Another influential aspect is *how* the unavailability of users is modelled in the corresponding MDP.

Introducing user availability into the MDP involves updating the transition `step` shown in Figure 2 to include setting the availability of the user, based on some given probability. Transition `step` is now read as:

- `step`: get the next workflow task t, a perspective user u to assign to t, and the availability of u.

The MDP arrives at state 2 with the next task and a perspective user to be assigned, who is either available of not available. As the MDP takes the transition `step` for each user, it follows availability must be set for all $u \in U$.

4.3 Availability Models

In [20], Wang and Li introduce three categories of resiliency, each using a different availability model: static resiliency, where users can only fail before the start of the workflow; decremental resiliency, where users can fail during the workflow, and cannot become available again; and dynamic resiliency, where users can fail and later become available during the workflow. These three resiliency categories form the basis of the work in [17] which predominately focuses on the decremental model.

Two main modelling approaches can be considered; the first simplistic approach consists of encapsulating the availability of all users for every task as a whole such that every user is given the same availability probability; the second, more complex approach consists of considering users individually such that different availability probabilities can be applied to each user. We focus on the latter approach unless stated otherwise.

We now consider a non-exhaustive number of ways a user's availability can be modelled, ranging from simple non-deterministic models to more complex models involving dependent user failure probabilities. Selecting which model to use may be dependent on the environment and user availability data such that the most accurate model possible can be chosen, or it could be a trade-off between high accuracy and fast computation. For ease of identification throughout the rest of this paper each availability model of interest is labelled am_i.

4.3.1 Non-deterministic Models

It may be the case that the availability of a user for a task is simply unknown, or non-deterministic. Therefore the availability of a user can be considered as binary, they are either available of not for a given user-task assignment. To effect a dynamic model (am_1), a user's availability is either *true* (available) or *false* (unavailable) for each task. In our running example the availability for u_3 may, for example be *true* for t_1, *false* for t_2 and *true* for t_3.

Recording the current availability of a user allows a decremental model to be established (am_2). If a user is unavailable for a task t they will remain unavailable for each remaining task t' in the workflow, where $t < t'$. In our running example, the availability for u_3 to be *true* for t_1, *false* for t_2 and *false* for t_3 corresponds to a valid decremental model. A static availability model (am_3) can be established by noting the current task. If no task has been selected (the workflow has not started), the availability of a user may be set as *true* or *false*. Once the first task t_1 has been selected, if the availability of a user is *true* it remains *true* for the entire workflow, if it is *false* it remains *false*. In our running example, the availability for u_3 to be *true* for t_1, *true* for t_2 and *true* for t_3 corresponds to a valid static model.

4.3.2 Probabilistic Models

It may be the case that some probability of availability can be applied to a user based, for example, on previous workflow logs and behavioural analysis. The most simple probabilistic model is to derive a single probability applied to every user for every task. However such an approach is likely to be impractical and provide little value.

A more complex variation (am_4) is to state a different probabilistic availability for each user which is applicable to all tasks in the workflow. In our running example, u_1 may have a probabilistic availability of 0.96 for t_1, t_2 and t_3, u_2 may have a probabilistic availability of 0.75 for t_1, t_2 and t_3, and so on. Another possibility (am_5) is to state a different probabilistic availability for each task in the workflow which is applicable to all users. This is equivalent to saying any user u has the probabilistic availability p for task t. In our running example, u_1, u_2 and u_3 may have a probabilistic availability of 0.98 for t_1, 0.78 for t_2 and 0.93 for t_3.

A further variation (am_6) is to apply a different probability for each user to be available for each task. In our running example, u_1 may have a probabilistic availability of 0.85 for t_1, 0.97 for t_2 and 0.64 for t_3, u_2 may have a probabilistic availability of 1 for t_1, 0.98 for t_2 and 0.86 for t_3, and so on. Notice how the probabilistic models introduced so far have independent probabilities. Note also that the models discussed in this section could be dynamic, decremental or static as described in Section 4.3.1.

We now consider more dependent probabilistic availability models. The first (am_7) considers the current availability of a user such that a user may have a different probability of availability if currently available to one if currently unavailable. In our running example, if u_3 was available for t_2 their probability of being available for the next task t_3 may be 0.98, whereas if u_3 was unavailable for t_2 their probability of being available for t_3 may only be 0.3.

Similarly, availability of a user u may depend on the current availability of another user u' (am_8) or a combination of both u and u' ($am9$). Many other more complex variations can also be imagined involving combinations of current user availabilities, current task and the historic availability of previous tasks.

4.3.3 Bounded Models

It could be desirable and more realistic allowing only a proportion of users to fail rather than allowing the unlikely possibility of every user failing. A bounded decremental model (am_{10}) introduces a threshold k to limit the maximum number of users failing during the operation of a workflow, similar to [20]. This means at least $|U| - k$ users will always be available for a task, although they may not be authorised to perform it. The threshold k may for example be the observed average number of users failing across all previous instances of a workflow.

Such a model must consider all possible failures of up to k users. For example, all k users may become unavailable at the same point or at different points in the workflow. Being decremental in nature, once a user is unavailable they remain so for the rest of the workflow. In our running example, imagine k=2 such that up to 2 users may become unavailable. User u_2 being available throughout the workflow whilst u_1 failing at t_2 and u_3 failing at t_3 would be one valid user failure case for am_{10}, as would u_1 being available throughout whilst u_2 and u_3 both fail at t_2. The cases where only one or even no users fail would be valid for k=2 whereas all three users failing would not.

A more dynamic model am_{11} can be defined by applying a threshold k to each task, in other words up to k users may become unavailable for a task independent of previous user failures. This model is dynamic in the sense that a user who has failed for task t may become available for the next task t'.

5. WORKFLOW MODELLING

There are many ways to solve an MDP including dynamic programming (e.g. value iteration) [13]. This technique is provided by the probabilistic model checking tool PRISM, which enables the specification, construction and analysis of probabilistic models such as MDPs [16]. PRISM is an intuitive choice as it can model both probabilistic and non-deterministic choice, and gives an efficient way to solve an MDP whilst providing analysis data regarding computation overheads.

5.1 Prism Modelling Language

A PRISM model definition constitutes a number of interactive modules that contain one or more local *variables* and *commands* as follows:

```
module name
    variables
    commands
endmodule
```

The values of a module's local variables define the module's state while the values of all variables across every module determines the global state of the model. Each variable is defined with a *name*, a *type* restricted to a finite range of integers or a Boolean, and an initial *value*:

```
name : type init value
```

The behaviour of a module is described through a set of local commands. Each command contains a *guard* and one or more probabilistic *updates*, taking the form:

```
[label] guard →prob_1 : update_1 & ... & prob_n : update_n;
```

A guard is a predicate over both local variables and those contained in other modules. If a command's guard equates to true, the guard's corresponding updates take place assigning one or more local variables with new values. The probability of $update_1$ is $prob_1$ and so forth such that the sum of $prob_1$ to $prob_n$ is 1. Only one of the updates will take place with its given probability assuming the guard is true. If a command contains only one update with probability 1, the probability value can be omitted. Updating variables is equivalent to a state transition causing the model to move from its current state to a new state. Given a variable `i` an update is defined in PRISM as `(i'=x)` where `x` is the new value assigned to `i`.

Labelling commands across modules with a common label allows those commands to be synchronised such that a transition consists of all these commands operating in parallel. Such a transition will only occur when the guards of all its constituent commands equate to true. A guard of the form **true** is an empty guard that always equates to true; commands with such a guard are used to update local variables regardless of their current value. These commands can still be synchronised with others through labelling. A module containing several commands whose guards all equate to true will make a non-deterministic choice over which command to perform:

```
[label_i] guard_i →update_1;
[label_i] guard_i →update_2;
```

Named expressions or *formulas* can be included in a PRISM model to avoid the duplication of code and reduce the state space. Essentially a formula's expression can be substituted by its name wherever the expression would be expected.

```
formula name = expression;
```

5.2 Workflow Encoding

We provide a PRISM encoding of our running example in Appendix A. Note we use the variable `t` to indicate the current task where `t=0` indicates the workflow has not started (no task has been selected), and `t=−1` indicates the workflow has terminated. The Boolean variable `fail` is set to **true** if a user violates the security policy or is unavailable, or **false** otherwise.

Probabilistic model verification can be performed automatically by PRISM to analyse properties defined in temporal logic. In terms of satisfiability we require PRISM to calculate the maximum probability that a workflow will terminate such that all tasks are assigned and all constraints satisfied. This satisfiability property is defined in PRISM as:

```
Pmax=? [ F (t=−1) & (!fail) ]
```

where: `Pmax =?` asks PRISM to find the maximal probability of the formula in between brackets to be true; the operator `F` is the "Eventually" operator, i.e., `F p` is true if the statement `p` eventually holds in the system. In this case, we ask PRISM to find the maximal property so that, eventually, the termination point has been reached, `(t=−1)` and the workflow has not failed, `(! fail )`. This probability is between 1 or 0, indicating the resiliency of the workflow.

5.3 Availability Encoding

In this section we examine PRISM's capability of modelling user availability by encoding the models am_i discussed in Section 4.3. Unless stated otherwise, user availability for each $u_i \in U$ is encapsulated in a distinct module named u_i. The availability of u_i for each task in the workflow is assigned by a PRISM command labelled `[s]`, synchronised to occur in the $[s]$ transition shown in Figure 2. If more than one `[s]` command is defined for each user, only one will execute dependant on its guard equating to *true*.

5.3.1 Non-deterministic Encodings

As the availability of a user u_i is considered a binary property, a single boolean f_i is set to **true** if u_i is available of **false** if not. The non-deterministic dynamic model am_1 can be encoded as follows:

```
module u_i
f_i : bool init true;

[s] true → (f_i'=true);
[s] true → (f_i'=false);
endmodule
```

To encode the non-deterministic decremental model am_2 we must ensure f_i remains **false** if it is already **false**, as follows:

```
module u_i
f_i : bool init true;

[s] f_i →(f_i'=true);
[s] f_i →(f_i'=false);
[s] !f_i →true;
endmodule
```

To encode the non-deterministic static model am_3 we must ensure f_i is only set before the first task (when `t=0`) and remains the same for every other task, as follows:

```
module u_i
f_i : bool init true;

[s] t=0 →(f_i'=true);
[s] t=0 →(f_i'=false);
[s] t!=0 →true;
endmodule
```

5.3.2 Probabilistic Encodings

To encode the probabilistic model am_4 only one availability probability p is used for u_i across all tasks. It follows that a single `[s]` command is required with an empty guard, as follows:

```
module u_i
f_i : bool init true;

[s] true → p:(f_i'=true) + 1 − p:(f_i'=false);
endmodule
```

In our running example, imagine u_2 has a probability of 0.86 of being available for any task in the workflow. The corresponding availability module for u_2 under model am_4 would be as follows:

```
module u2
f2 : bool init true;

[s] true →  0.86:( f2'=true) + 0.14:(f2'=false);
endmodule
```

The probabilistic models am_5 and am_6 can be encoded in the same way for each user u_i, the difference is in the applied probabilities. In am_5 the probabilistic availability p_i for t_i will be the same for all $u \in U$ ($|T|$ probabilities) whilst in am_6 they may be different (up to $|U| \times |T|$ probabilities). In order to set a probabilistic availability before each task t_i is assigned (t_n being the last task), a separate command [s] is given for each t_i, as follows:

```
module u_i
f_i : bool init true;

[s] t=0 → p:(f_i'=true) + 1 − p:(f_i'=false);
⋮
[s] t=n − 1 →p_n:(f_i'=true) + 1 − p_n:(f_i'=false);
endmodule
```

The dependant probabilistic model am_7 is encoded to allow for a different probabilistic availability to be set for user u_i according to the current availability of u_i. This is effected with two [s] commands, the first executes if u_i is currently available (f$_i$=**true**), the second executes otherwise. Note how the number of commands could be increased with more precise guards, such that the described behaviour can be implemented for each individual task.

```
module u_i
f_i : bool init true;

[s] f_i → p:(f_i'=true) + 1 − p:(f_i'=false);
[s] !f_i → p':(f_i'=true) + 1 − p':(f_i'=false);
endmodule
```

The dependant probabilistic model am_8 allows for a different probabilistic availability to be set for user u_i according to the current availability of another user u_j. PRISM allows modules to read the status of variables in other modules (but not update them). In our encoding f$_j$ denotes the current availability of user u_j.

```
module u_i
f_i : bool init true;

[s] f_j → p:(f_i'=true) + 1 − p:(f_i'=false);
[s] !f_j → p':(f_i'=true) + 1 − p':(f_i'=false);
endmodule
```

The dependant probabilistic model am_9 is essentially a combination of am_7 and am_8, encoded as follows:

```
module u_i
f_i : bool init true;

[s] f_i & f_j →p:(f_i'=true) + 1 − p:(f_i'=false);
[s] !f_i & !f_j → p':(f_i'=true) + 1 − p':(f_i'=false);
endmodule
```

5.3.3 Combined Encoding

PRISM's state-based language is powerful enough to model both non-deterministic and probabilistic user availability, allowing us to combine both into a single encoding. In our running example, imagine for u_2 we have no foresight of their availability for t_1 (non-deterministic), we predict a probabilistic availability of 0.86 for t_2, we predict a probabilistic availability of 0.95 for t_3 if u_2 was available for t_2, and a probabilistic availability of 0.65 for t_3 if u_2 was not available (failed) for t_2. To clarify, the command [s] with the guard t=0 is setting the availability of u_2 for the next task t_1, t=1 for the next task t_2, and so forth. An encoding of this would be as follows:

```
module u2
f2 : bool init true;

[s] t=0 →(f2'=true);
[s] t=0 →(f2'=false);
[s] t=1 →  0.86:( f2'=true) + 0.14:(f2'=false);
[s] t=2 & f2 →  0.95:( f2'=true) + 0.05:(f2'=false);
[s] t=2 & !f2 →  0.65:( f2'=true) + 0.35:(f2'=false);
endmodule
```

5.3.4 Bounded Encodings

It is not trivial to encode non-deterministic bounded failures in PRISM. Encoding the step [s] in Figure 2 involves synchronising all commands assigning the availability of all users, in other words all these commands execute in parallel. It is not possible to randomly limit the number of executed commands to k or $< k$ if some users have already failed. For instance, in our running example we require at least 3 [s] commands to assign their availability. If we want only 2 at most users failing (i.e. k=2) it is not possible to allow either 1 or 2 commands to be executed in a non-deterministic way.

To encode am_{10} we explicitly state every possible user failure case within a single PRISM command [u] and assign a probability to each case such that their total is 1. The same decremental model is used in [17] where equal probabilities for each user failure case are applied. For simplicity we also consider an equiprobable model such that every combination of failures of up to k users has an equal chance of occurring. This means all failure cases must be evaluated when calculating workflow resiliency. In our encoding, the command [u] assigns all user availabilities before the first workflow task. This command can be imagined to be equivalent to a new transition labelled *users*, from a new state 5 to the existing state 1 shown in Figure 2.

A variable f$_i$ initialised to 0, holds the step at which user u_i fails. The value t$_n$+1 indicates a user does not fail where n is the number of tasks in the workflow. For example, in our running example, the variable f1=4 can be read either as u_1 does not fail or u_1 does fail but at some point after the last task t_3, which is of no consequence. Each possible failure case is represented as a vector which includes every user and their failure step. For instance, in the running example u_1 failing at t_2, u_2 failing at t_3 and user u_3 not failing constitutes a single failure vector which is defined in PRISM as a distinct probabilistic update, where q is the number of all possible failure vectors:

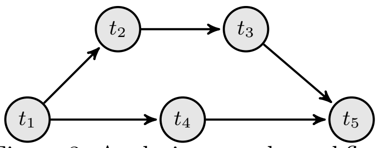

Figure 3: Analysis example workflow

```
| 1/q:(f1'=2) & (f2'=3) & (f3'=4)
```

The command [u] combines all probabilistic updates to make an equiprobable choice over which failure vector to choose before updating each variable f_i accordingly:

```
| [u] true → 1/q : update_1 + ... + 1/q : update_q;
```

In our running example, imagine we consider all failures of up to 2 users ($k = 2$), creating 37 failure vectors. A single module availability is used to capture all user availability as follows:

```
| module availability
| f1 : [0..4] init 0;
| f2 : [0..4] init 0;
| f3 : [0..4] init 0;
|
| [u] true → 1/37:(f1'=1)&(f2'=4)&(f3'=4)
|     + ... + 1/37:(f1'=4)&(f2'=4)&(f3'=4);
| endmodule
```

To assess the availability of the perspective user pu for a particular task t, the formula av is added to the model definition as follows:

```
| formula av = (pu=1 & t<f1) | ... | (pu=n & t<f_n);
```

For example, if the failure step of u_1 is t_3 (f1=3) it follows (pu=1 & t<f1) will be true when t=1 and t=2, in other words u_1 is available up until t_3. A further implementation change to encode am_{10} is to update the main module of the model definition given in Appendix A. This includes inclusion of a variable uf and command [u] as shown in Appendix B.

The more dynamic bounded model am_{11} can be encoded by simply changing the command label from [u] to [s]. The availability of each user is now assigned with up to k users failing for each task in the transition [s] in Figure 2. Note the state 5 and transition *users* required for am_{10} is not required, nor are the changes to the main module given in Appendix B.

6. ANALYSIS

In this section we look at how the different user availability models previously described can affect resiliency computation, not only in terms of value but also in terms of complexity including model state space, memory and CPU time.

For our analysis we consider a larger workflow example shown in Figure 3 where $T = \{t_1, t_2, t_3, t_4, t_5\}$ and $t_1 < t_2 < t_3 < t_5$ and $t_1 < t_4 < t_5$. It follows that three valid task execution sequences exist, $\langle t_1, t_2, t_3, t_4, t_5\rangle$, $\langle t_1, t_2, t_4, t_3, t_5\rangle$, and $\langle t_1, t_4, t_2, t_3, t_5\rangle$. The workflow also comes with users $U = \{u_1, u_2, u_3, u_4\}$, and a security policy $p_2 = (P_2, S_2, B_2)$ where:

- $P_2 = \{(u_1, t_1), (u_2, t_1), (u_2, t_2), (u_3, t_2), (u_1, t_3), (u_2, t_3), (u_2, t_4), (u_4, t_4), (u_1, t_5), (u_4, t_5)\}$
- $S_2 = \{(t_2, t_4), (t_3, t_4), (t_4, t_5)\}$
- $B_2 = \{(t_1, t_3)\}$

We select for analysis the following availability models described in Section 4.3 and encoded in Section 5.3:

- am_1 : dynamic, non-deterministic
- am_4 : dynamic, independent, probabilistic (simple)
- am_6 : dynamic, independent, probabilistic (complex)
- am_7 : dynamic, dependent, probabilistic
- am_{10} : decremental, bounded (k=2), probabilistic
- am_{11} : dynamic, bounded (k=2), probabilistic

Each availability model encoding is used in turn with the workflow encoding given in Appendix A. The workflow resiliency is computed in PRISM using its default *hybrid* engine[1] run on a computing platform incorporating a 2.40Ghz i7-4500U Intel processor and 8GB RAM. The results generated are given in Table 1.

6.1 Resiliency Analysis

Clearly changing the probability of a user being available is going to change the overall resiliency value calculated for a workflow. This applies to any probabilistic user availability model making an exact comparison in terms of the calculated resiliency value challenging. The availability models also vary in complexity, for example am_1 is purely non-deterministic, am_4 considers 4 probabilities, whilst am_6 considers 20 probabilities.

Instead we place an arbitrary bounds of between 0.7 and 1 on the randomly generated probabilities for each availability model. This experiment is run 50 times per model and the averages taken. Bounding the probabilities ensures each model is using roughly similar values. Even with this check in place, the resiliency results in Table 1 exhibit a wide variation of between 100 and 17.65%. We do not suggest which is the best model to use for resiliency calculation but instead highlight that different model choices can impact the resulting value for the same workflow.

With the non-deterministic model am_1 there will always be the possibility of a valid user being available for each task, thus a 100% resiliency value is achieved. This is arguably of little value in terms of resiliency analysis. However in terms of the WSP, it indicates the workflow is *satisfiable.*

6.2 Resiliency Complexity

In this section we compare resiliency complexity given by different availability models in terms of state space, computation time and memory. Such metrics are useful especially if runtime resiliency analysis is required. Of course, complexity metrics of this type will be somewhat dependent on the computing platform and model checker used. The complexity results given in Table 1 are placed under the following columns:

- States - total number of reachable states in the PRISM model
- Transitions - total number of transitions between states
- Build time - time to construct the model
- Verify time - time to compute resiliency
- Memory - total memory used to compute resiliency
- File size - size of PRISM model file
- Size on disk - amount of storage allocated by the OS for the PRISM file.

In Table 1 we can see each of the unbounded availability models, am_1, am_4, am_6 and am_7 all result in the same sized model, i.e., same number of reachable states and number of transitions between states. It follows that they all have

[1]http://www.prismmodelchecker.org/manual/

Table 1: Analysis result averages

	resiliency (%)	states	transitions	build time (s)	verify time (s)	memory (MB)	file size (KB)	size on disk (KB)
am_1	100.00	8530	31321	0.219	0.015	-	2.51	4.00
am_4	40.75	8530	31321	0.172	0.016	-	2.50	4.00
am_6	78.76	8530	31321	0.172	0.016	-	3.21	4.00
am_7	71.71	8530	31321	0.156	0.016	-	2.67	4.00
am_{10}	42.69	50489	64377	0.125	0.172	1.3	8.95	12.00
am_{11}	17.65	91232	3949682	0.062	0.359	17.5	8.90	12.00

similar build and verification times. An obvious observation is that a model complex to construct with many probabilities such as am_6, has the same computational complexity as a much simpler model, e.g. am_1. In other words the model size will remain constant if the workflow and security policy are not changed.

One noticeable observation is the difference in file size which increases as more content (e.g. commands, probabilities, etc) is added to the model as one would expect. For example, am_1 results in a file size of 2.51KB whilst the larger model am_6 results in a file of 3.21KB for this example. Despite this the actual storage space allocated by the OS for all four unbounded model files is the same at 4KB. Therefore a simple model may use the same storage space as a more detailed availability model.

The bounded availability models am_{10} and am_{11} result in a large increase in the overall workflow model complexity. For example, the state space using am_{11} is comprised of 91232 states compared with 8530 states when using an unbounded availability model. Similarly, the number of transitions using am_{11} is 3949682 compared with 31321. This can be attributed to the explicit encoding of all possible user failure cases in these bounded availability models. The corresponding computation times, memory usage and storage size are also observed to increase in-line with the size of the workflow model.

7. CONCLUSION

We have reused the result in [17] which considers the *workflow satisfiability problem* as a decision problem, reduced to that of solving a Markov Decision Process (MDP). We have provided an encoding of workflows as probabilistic models and used the ability of the model checker PRISM to efficiently solve the resulting MDP. The *workflow resiliency problem* is also automatically solved following the introduction of several user availability models.

We have shown that user availability can be modelled in several ways for the same workflow, set of users and security policy making the correct choice of model a complex yet necessary concern due to its major impact on the computed resiliency value. It is important that a workflow security policy designer is provided with suitable values when assessing how resiliency would be impacted by a security policy. Appropriate resiliency values are also useful to ensure a policy provides the minimum requirement of resiliency; highlighting extra user level requirements, unworkable processes and policies; and in providing resiliency indicators and assurance to business leaders and process designers. We also show how model choice can impact resiliency computation in terms of memory and CPU time.

In terms of future work we look to analyse different sizes of workflow to understand how the complexity of computing resiliency scales and how the complexity metrics change. We also look to develop suitable tools and methodologies for workflow security policy designers enabling them to automatically calculate an appropriate resiliency value offered by a workflow security policy.

8. REFERENCES

[1] Workflow handbook 1997. chapter The Workflow Reference Model, pages 243–293. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[2] A. Armando and S. E. Ponta. Model checking authorization requirements in business processes. *Computers & Security*, 40(0):1 – 22, 2014.

[3] V. Atluri and J. Warner. Security for workflow systems. In M. Gertz and S. Jajodia, editors, *Handbook of Database Security*, pages 213–230. Springer US, 2008.

[4] D. Basin, S. J. Burri, and G. Karjoth. Obstruction-free authorization enforcement: Aligning security with business objectives. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 99–113, Washington, DC, USA, 2011. IEEE Computer Society.

[5] A. Basu and A. Kumar. Research commentary: Workflow management issues in e-business. *Info. Sys. Research*, 13(1):1–14, Mar. 2002.

[6] R. Bellman. A markovian decision process. *Indiana Univ. Math. J.*, 6:679–684, 1957.

[7] R. Botha and J. H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Systems Journal*, 40(3):666–682, 2001.

[8] J. Crampton, G. Gutin, and A. Yeo. On the parameterized complexity and kernelization of the workflow satisfiability problem. *ACM Trans. Inf. Syst. Secur.*, 16(1):4, 2013.

[9] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and parallel Databases*, 3(2):119–153, 1995.

[10] L. He, C. Huang, K. Duan, K. Li, H. Chen, J. Sun, and S. A. Jarvis. Modeling and analyzing the impact of authorization on workflow executions. *Future Generation Computer Systems*, 28(8):1177–1193, 2012.

[11] L. Herbert and R. Sharp. Precise quantitative analysis of probabilistic business process model and notation workflows. *Journal of Computing and Information Science in Engineering*, 13(1):011007, 2013.

[12] H. Hiden, S. Woodman, P. Watson, and J. Cala. Developing cloud applications using the e-science

central platform. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1983):20120085, 2013.

[13] R. A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA, 1960.

[14] M. Kohler, C. Liesegang, and A. Schaad. Classification model for access control constraints. In *Performance, Computing, and Communications Conference, 2007. IPCCC 2007. IEEE Internationa*, pages 410–417, April 2007.

[15] M. Kohler and A. Schaad. Avoiding policy-based deadlocks in business processes. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 709–716, 2008.

[16] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[17] J. Mace, C. Morisset, and A. van Moorsel. Quantitative workflow resiliency. In *Computer Security - ESORICS 2014*, volume 8712 of *Lecture Notes in Computer Science*, pages 344–361. Springer International Publishing, 2014.

[18] F. Martinelli and C. Morisset. Quantitative access control with partially-observable markov decision processes. In *Proceedings of*, CODASPY '12, pages 169–180, New York, NY, USA, 2012. ACM.

[19] J. F. Meyer and W. H. Sanders. Specification and construction of performability models. In *Proceedings of the Second International Workshop on Performability Modeling of Computer and Communication Systems*, pages 28–30, 1993.

[20] Q. Wang and N. Li. Satisfiability and resiliency in workflow authorization systems. *ACM Trans. Inf. Syst. Secur.*, 13(4):40:1–40:35, Dec. 2010.

APPENDIX

A. WORKFLOW ENCODING

Here we provide a PRISM encoding of our running example. Example encodings for the user availability modules u1, u2 and u3 are provided in Section 5.3.

```
module main
nt : bool init true;
pc : bool init false;
fail : bool init false;

[s] nt → (nt'=false)&(pc'=false);
[p] !nt & !pc → (pc'=true);
[a] !nt & pc & pol → (nt'=true);
[e] nt → true;
[f] !nt & pc & !pol & ! fail → ( fail '=true);
endmodule

//task selection
module workflow
t : [−1..3] init 0;
t1 : bool init false;
t2 : bool init false;
t3 : bool init false;

[s] !t1 → (t'=1)&(t1'=true);
[s] t1 & !t2 → (t'=2)&(t2'=true);
[s] t2 & !t3 → (t'=3)&(t3'=true);
[s] t3 → (t'=−1);
[f] true → (t'=−1);
endmodule

//user selection
module candidate
pu : [0..3] init 0;

[s] true → (pu'=1);
[s] true → (pu'=2);
[s] true → (pu'=3);
endmodule

//security policy
formula pol = perms & sods & av;

//permissions
formula p1 = t=1 & (pu=1 | pu=2);
formula p2 = t=2 & (pu=2 | pu=3);
formula p3 = t=3 & (pu=1 | pu=3);
formula perms = p1 | p2 | p3;

//separation of duties
module sod1
us1 : [0..3] init 0;
fs1 : bool init false;

[p] (t=1 | t=3) & us1=0 →(us1'=pu);
[p] (t=1 | t=3) & us1!=0 & pu=us1 →(fs1'=true);
[p] (t!=1 & t!=3) | (us1!=0 & pu!=us1) →true;
endmodule

module sod2
us2 : [0..3] init 0;
fs2 : bool init false;

[p] (t=2 | t=3) & us2=0 →(us2'=pu);
[p] (t=2 | t=3) & us2!=0 & pu=us2 →(fs2'=true);
[p] (t!=2 & t!=3) | (us2!=0 & pu!=us2) →true;
endmodule

formula sods = !fs1 & !fs2;

// availability
module u1
f1 : bool init true;

...
endmodule

module u2
f2 : bool init true;

...
endmodule

module u3
f3 : bool init true;

...
endmodule

formula av = (pu=1 & f1) | (pu=2 & f2) | (pu=3 & f3);
```

B. BOUNDED ENCODING

Here we provide a PRISM encoding of the main module when using the bounded user availability module described in Section 5.3.4.

```
module main
uf : bool init false ;
nt : bool init true;
pc : bool init false ;
 fail : bool init false ;

[u] !uf → (uf′=true);
[s] uf & nt → (nt'=false)&(pc'=false);
[p] !nt & !pc → (pc'=true);
[a] !nt & pc & pol → (nt'=true);
[e] nt → true;
[f] !nt & pc & !pol & ! fail → ( fail '=true);
endmodule
```