ProActive Caching: Generating Caching Heuristics for Business Process Environments

Mathias Kohler
SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe
Germany
Email: mathias.kohler@sap.com

Achim D. Brucker
SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe
Germany
Email: achim.brucker@sap.com

Andreas Schaad
SAP Research
Vincenz-Priessnitz-Str. 1
76131 Karlsruhe
Germany
Email: andreas.schaad@sap.com

***Abstract*—Today's complex and multi-layered enterprise systems demand fine-grained access control mechanisms supporting dynamic security policies for large and distributed repositories. Thus, the efficient evaluation of security policies becomes an important factor for the overall system performance, specifically with respect to systems with a high degree of user interaction like workflow systems. Caching approaches may help to address this situation.**

We propose ProActive Caching, a two-phased caching approach: in an *offline phase*, we automatically determine a workflow-specific heuristic for pre-computing cache entries. In an *online phase*, we use the previously determined heuristic for the cache management. The latter includes also the pre-computation of cache entries which already provides a performance improvement while evaluating a policy object for the first time. In this paper, we present a method for the automatic generation of a workflow specific caching heuristic, i. e., the offline phase.

I. Introduction

Large distributed enterprise systems require the implementation of mechanisms for the enforcement of fine-grained, complex and dynamic access control policies. Access control decisions need to be made at different layers, such as the UI layer, the application layer, the business object layer as well as the back-end or service layer. Equally, on a vertical level, business requirements driven by, e. g., regulatory demands, result in an increase in the amount and complexity of policies, including the need for checking the current system state, and an increase in the number of policy evaluation events. This, of course, affects policy evaluation at runtime and thus, the overall system performance experienced by the user.

The enforcement of such policies should be invisible and not noticeable to the interacting user. Practical experience shows that even delays of $100\,\mathrm{ms}$ are perceived as a minor interruption, while more than 1 second impacts on the flow of thoughts [13]. State of the art industrial workflow systems execute hundreds of thousands of business process tasks in parallel, making the efficient implementation of policy enforcement and evaluation techniques increasingly important.

State of the art security infrastructures are implemented with efficiency in mind [10], [15] and, to some extend, already apply caching strategies [11], [20]. All these approaches have in common that the presented optimization techniques are generic, i. e., trying to optimize access control decisions independently from the system context. In [9], we report on our first experience in applying caching strategies that are based on high-level system descriptions, e. g., business processes. This integration of information about the possible system behavior allows to anticipate future access control decisions and thus provides an efficient caching solution that increases the overall system performance substantially, even if the system is already using a highly efficient security infrastructure.

Our main contribution is a caching strategy that, based on high-level system descriptions, is not affected by cache misses and already accelerates the first access to a resource. In particular, we present an algorithm for the automatic generation of cache update heuristics based on given process definitions and life cycles (for tasks and processes) of an underlying workflow engine. We present an analysis of our overall approach with respect to its runtime behavior.

II. Background

In this section, we introduce business process-driven systems and proactive caching of access control decisions.

A. Business Process Execution

Business processes "focus upon the production of particular products" [18] realized by a sequence of tasks. Processes may be modeled in total or in parts as workflows. Figure 1 shows a simple *process model*: the nodes represent the different *tasks*, i. e., atomic activities. The transitions describe the execution order of the tasks. In our example the process flow splits, i. e., Task 1 and Task 2 can be executed in parallel. Finally, source and sink nodes represent the starting point and endpoint of a process model. A process model can be constrained by a security policy. To illustrate that roles have the permission to execute an activity we add a small circled annotation to every activity. Dynamic policies, e. g., separation of duty (SoD) can also be specified: in our example, we require that Task 1 and Task 2, and, pairwise, Task 2 and Task 3, must be performed by different users.

A workflow engine, e. g., JBoss jBPM [17], executes processes based on their abstract model. In particular, a workflow engine creates, executes, and deletes instances of processes and

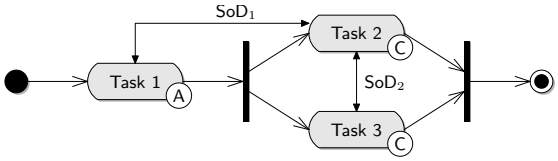


Fig. 1. A Simple Business Process Model.

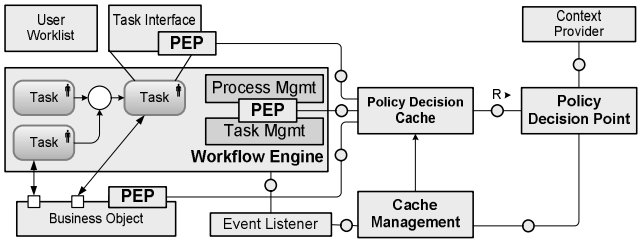


Fig. 2. Business Process Execution Layers.

tasks with respect to life cycles which describe the different technical states of the process and task representations.

Human centric execution of business processes requires a human to execute the tasks. A user claims a task, gets assigned to it and starts executing it by means of interacting with the business process application. All tasks a user can claim are listed in the General Worklist (GWL). The user opens the GWL and claims the task(s) she wants to perform next. The GWL should only display those tasks a user is allowed to claim (usually only a small fraction of the total number of active tasks). Thus, for each task instance in the system a potential list of owners is generated. Assuming role-based access control (RBAC) [14], this list is created based on the roles a user must possess to perform the task. Due to dynamic constraints, this list is an over-approximation, i. e., even potential owners may be not allowed to claim a task in a specific situation. Whenever the GWL is displayed, a second access control query checks for every task instance in the list whether it can be claimed.

In [9] we show that ProActive Caching effectively reduces the time required for access control requests. For instance, the GWL can be displayed without a noticeable delay caused by evaluating access control decisions. To gain this performance increase it is important that the access decisions are already available when the worklist is opened. In this paper we introduce the automatic generation of the underlying caching heuristic that guarantees that access decisions—which are required for displaying the GWL—are already cached.

B. ProActive Caching

The goal of proactive caching is to decrease the overall response time experienced by the user. The core idea is the prediction of future access queries allowing to pre-compute the required cache entries. In [9], we introduce the basic underlying framework and show that that proactive caching substantially reduces response times of the systems.

In large enterprises systems access control is usually based on the request-response paradigm. This describes the interaction between a policy enforcement point (PEP) and a policy decision point (PDP). The PEP is part of the application, enforcing the decisions made by the PDP. Figure 2 presents such a generic access control architecture extended with an access decision cache and a cache management component. Each cache entry is a pre-evaluated access decision which is anticipated to be required during the further execution of a business process. The cache management component is responsible that access decisions are pre-evaluated and stored in the cache. Thus, the management component relies on generated caching heuristics which define at which point during a process execution access decisions must be pre-evaluated. The current status of the process execution is communicated via events to the cache management. Access pre-evaluations are performed by the PDP. When triggered by an event, the cache management component sends the access request for which an access decision should be pre-evaluated to the PDP. The PDP evaluates the request and returns a corresponding access decision which is stored in the cache.

C. Caching Heuristics

The pre-evaluations of access control decisions is based on caching heuristics, in particular *dependency relations* (DR). A DR maps a pair (event, resource) to a pair (event, resource), i. e., its type is $\Sigma \times R \rightarrow \Sigma \times R$ where Σ is the set of events and R the set of resources. The event (and the corresponding resource) at which an anticipated access request should be evaluated is mapped to the event-resource-pair for which an access decision should be pre-compute. For example, the dependency relation

$$(\text{createProcess, 'P'}) \mapsto (\text{cancelProcess, 'P'})$$

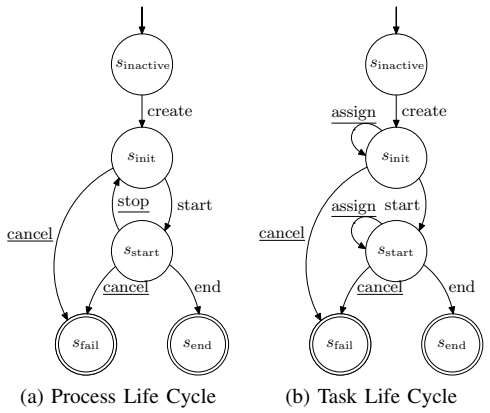
states that whenever the event 'createProcess' for the resource 'P' occurs an access request for the anticipated upcoming event 'cancelProcess' (with resource 'P') will be evaluated such that the access decision is already available in case the process should be canceled. During runtime every event occurring in the system is matched against the events defined in the dependency relations. As soon as an event matches, an access control request is sent to the PDP based on the second event defined in the relation to pre-compute the access control decision. The returned decision is stored as a cache entry.

III. Dependency Relations for Business Processes

In this section, we introduce the sources life cycles, process definitions, business objects, and SoD constraints from which dependency relations can be generated.

A. Relations Based on Life Cycles

At runtime, every process and task instance may potentially be in one of several states. A life cycle may comprise the following states: Inactive, Initiated, Started, Failed, and Terminated. Every vendor of workflow engines may define its own life cycles. Every state may only be reached through certain transitions. Such transitions reflect events performed on an instance. For example, the transition start is the event when an (initiated) instance is started. We model life cycles as deterministic finite state machines [6]. Figure 3 presents the



(a) Process Life Cycle (b) Task Life Cycle

Fig. 3. The Life Cycles of JBoss jBPM in their Formal Representations.

life-cycle models used by the JBoss jBPM workflow engine. Formally, we define a life cycle as follows:

Definition 1: A *formal life cycle model*, represented as finite state machine, is a quintuple $(Q, \Sigma, q_0, \delta, F)$, where:

- Q is a finite, non-empty set of states.
- Σ is a finite, non-empty set of events with $Q \cap \Sigma = \emptyset$.
- q_0 is the *initial state*, with $q_0 \in Q$.
- δ is the generalized state-transition function: $\delta : Q \times \Sigma \rightarrow Q$ describing the possible transitions.
- F is the set of *final states* with $F \subseteq Q$.

Further, we make this definition access control aware:

Definition 2: A caching and *access control aware formal life cycle model* is a sextuple $(Q, \Sigma, \Sigma_{\text{AC}}, q_0, \delta, F)$, where $(Q, \Sigma, q_0, \delta, F)$ is a life cycle and $\Sigma_{\text{AC}} \subseteq \Sigma$ is a finite set of *access control relevant* events.

For our exemplary task life cycle (see Figure 3b) this results in a formal life cycle model $(Q, \Sigma, \Sigma_{\text{AC}}, q_0, \delta, F)$, where

- $Q = \{s_{\text{inactive}}, s_{\text{init}}, s_{\text{start}}, s_{\text{fail}}, s_{\text{end}}\}$,
- $\Sigma = \{\text{create}, \text{assign}, \text{start}, \text{cancel}, \text{end}\}$,
- $\Sigma_{\text{AC}} = \{\text{assign}, \text{cancel}\}$,
- $q_0 = s_{\text{inactive}}$,
- $\delta = \big\{(s_{\text{inactive}}, \text{create}) \mapsto s_{\text{init}}, (s_{\text{init}}, \text{assign}) \mapsto s_{\text{init}}, (s_{\text{init}}, \text{start}) \mapsto s_{\text{start}}, (s_{\text{init}}, \text{cancel}) \mapsto s_{\text{fail}}, (s_{\text{start}}, \text{assign}) \mapsto s_{\text{start}}, (s_{\text{start}}, \text{end}) \mapsto s_{\text{end}}, (s_{\text{start}}, \text{cancel}) \mapsto s_{\text{fail}}\big\}$, and $F = \{s_{\text{fail}}, s_{\text{end}}\}$.

In Figure 3b, the initial state is denoted with an incoming arrow without starting point, end states are marked with a double circle, and access control relevant events are underlined. The classification of access control relevant events depends on the concrete implementation of workflow engine. Usually, events which require user input underlie access control enforcement whereas automatically executed events run without authorization checks. For example, the transition *end* in the process life cycle is executed automatically after the termination of all task instances of the process.

Σ_{AC} contains the events for access control decisions should be pre-computed. The pre-computation should take place whenever a predecessor event of an access control relevant even occurs, i. e., every state that is source of an access control relevant event is also the destination of events which precede the access control relevant event.

B. Relations Based on Process Definitions

Access control relevant events in task life cycles can occur directly after a task instance was created and hence must be checked against the systems security. The goal is to guarantee that directly after the creation of a task instance, cached decisions for possibly immediately following access control relevant events are provided. An example is the access control relevant event *assign*. Hence, instead of using the task's create event as trigger for the creation of an anticipated access decision, we use the create event of the preceding task in the process to pre-evaluate the access decisions, e. g.,

$$(\text{init event}, \text{`preceding task of Task N'}) \mapsto (\text{access control relevant event}, \text{`Task N'}),$$

where 'init event' is a placeholder for all events having s_{inactive} as source state (i. e., 'create' in our case). If a task has no preceding tasks (i. e., it is the first task of a process), the 'init event' of the process life cycle is used as trigger instead

$$(\text{init event}, \text{`P'}) \mapsto (\text{access control relevant event}, \text{`Task 1'}),$$

where 'P' is the process as resource. As every process definition exactly states in which order tasks are executed, one can extract all tasks which potentially succeed the current one.

C. Relations Based on SoD Constraints

Access control often requires context information for the evaluation of security constraints. The evaluation of time based constraints (e. g., a user may only perform a task within 6 a.m. and 7 p.m.) requires the current time to evaluate whether access should be granted or denied. We classify such context information which underlies steady or unexpected changes as *dynamic context information*. Cached decisions which are based on dynamic context information possibly become invalid as the context changes unnoticed (which we handle in previous work with open constraints; cf. [9]).

There are constraint types where the context information only changes within the boundary of the system's infrastructure such that whenever the context changes an update of possibly changing cached decisions can be performed. An example of a security constraint where context information only changes due to workflow related actions (i. e., actions part of a process or task life cycle) is dynamic separation of duty (SoD). Dynamic SoD demands that a user may not be assigned to two exclusive tasks of the same process instance.

Definition 3: A *separation of duty constraint* is a triple $(\mathit{Process}, \mathit{Event}, \mathit{exclTasks})$ where

- *Process* is the process for which the constraint is defined.
- *Event* is an access control relevant event of a task life cycle; when occurring in context of *Process* this SoD constraint is checked.
- *exclTasks* is a set of tasks of the *Process* with the implicit condition that a user may be assigned to at most one of these tasks within the same process instance.

Consider the exclusive tasks Task 2 and Task 3 in our example process during an execution where Task 2 and Task 3 are

ready to be assigned to users. For both tasks cache entries with access decisions have been pre-computed. The cached decision for Task 2 states that Alice may be assigned to it; the decision for Task 3 also states Alice may be assigned to it. Both permits result–due to the fact that at the point in time when the cached decisions were evaluated–Alice would have had the right to be assigned to Task 2 and Task 3. If Alice claims Task 2, according to the security policy she may not claim Task 3 as well—which should also be reflected by the pre-evaluated cached access decision. Hence, an update of the cache entry for Task 3 is necessary.

The goal is to generate dependency relations based on the SoD constraints defined in a security policy. We generate the dependency relation in such a way that if an event changes the context information and this affects an already pre-computed access decision, a new decision is computed which updates the previously generated one. For illustration, the dependency relations for the process of Figure 1 state that if a user is assigned to Task 2 the cached decision for the event (assign, 'Task 3') must be calculated anew. The same holds for the event (assign, 'Task 2') if a users is assigned to Task 3.

$$(\text{assign, 'Task 2'}) \mapsto (\text{assign, 'Task 3'}), \text{and}$$
$$(\text{assign, 'Task 3'}) \mapsto (\text{assign, 'Task 2'}).$$

D. Relations Based on Business Object Calls

Business Objects provide the functionality for data queries and data modifications on the back-end layer. Access control also happens for function calls on business objects. It is checked whether the user may perform the function called.

During a process execution, tasks access business object functionality. Which objects and what functions are accessed by a task is defined within the process definition. Hence, when as user is assigned to a task, it is clear that this user will access the business object. Consequently, we can pre-evaluate access decisions for these function calls at the moment we know which user is assigned to a task. In JBoss jBPM this is known when the event 'assign' occurs as this event commits a user to perform a task. The dependency relations for a JBoss jBPM system define the event 'assign' (and the respective task as resource) as the trigger for the pre-computation of all access decisions for the business object functions which will be called during execution of the task. A general example is given next.

$$(\text{assign, 'Task'}) \mapsto (\text{function call, 'Business Object'})$$

Due to space reasons, we omit the generation of dependency relations for business object calls in this paper. Due to the fact that there is a specific link between task definition and business object functions, the modification of the algorithm to include business calls is only a minor modification.

E. Revoke Trigger

Once a pre-computed access decision is cached it should be revoked from the cache as soon as it is not needed any longer. Similar to dependency relations there are *revoke triggers* which define the concrete point during a process execution at which cache entries are to be removed from the cache.

Decisions are always pre-computed for a specific instance of a task. Hence, whenever a task instance is completed all cache entries generated for this instance can be removed. The same holds for cache entries which were generated in context of a process instance. If the process instance ends all cache entries generated in its context can be removed.

Obviously, cache entries need to contain the information for which process or task instance they were created. This information is stored with the cache entry itself. This enables a direct selection of all cache entries which belong to a specific process or task instance. A revoke trigger contains the event at which all entries for either a task or process instance must be revoked. The events for a process are those which end a process instance, hence, which lead to an end state in the process life cycle. The events for a task are those which end a task instance. The set of revoke triggers for our process life cycle (PLC) Figure 3a is

$$\{(\text{end, 'Process'}), (\text{cancel, 'Process'})\},$$

where end, and cancel are those events which lead to the end states of the PLC. Events from the system contain the identifier for which instance the event happened. If the process instance for a process Process is canceled, an event with $(\mathsf{cancel}, \mathsf{Process}, 4711)$ is generated. If such an event matches with a revoke trigger all cache entries containing the respective instance identifier 4711 are revoked from the cache.

IV. Automated DR Generation

In this section, we present algorithms for computing the different dependency relations (DR) based on the life cycles and the process definition. Furthermore, we present an algorithm for DRs based on the system's security policy and an algorithm for the generation of revoke triggers.

A. Preliminaries

In our presentation, we roughly follow the pseudocode conventions of [3], e. g.,

- the symbol _ ← _ denotes variable assignments.
- variables are local to the given function.
- compound data, e. g., life cycles, are organized in objects or structures, which are comprised attributes or fields. For example, we write $\mathit{out}[n]$ ($\mathit{in}[n]$) for accessing the set of outgoing (incoming) events of state n and $\mathit{src}[e]$ ($\mathit{dest}[e]$) for accessing the source (destination) state of event e.
- we re-use the notations from Definition 2, e. g., $Q(\mathit{TLC})$ refers to the states of the task life cycle *TLC*; $F(\mathit{TLC})$ refers to the end states of a task life cycle.
- parameters are passed to a function by value.

We assume that life cycles and process definitions are stored in data structures which allows for efficient access to the states (tasks) and events. In particular, we assume that we can iterate over all states (or tasks) of a life cycle (or process), access the incoming and outgoing transition of a given state directly, and that we can access the source state of a transition.

```
1: function GenDR-LC(LC, res, event, initRes)
2:     DrSet_LC ← ∅
3:     for all s ∈ Q(LC) do
4:         for all i ∈ in[s] do
5:             for all o ∈ out[s] ∩ Σ_AC(LC) do
6:                 if src[i] = q_0(LC) then
7:                     entry ← (iEvent, iRes) ↦ (o, res)
8:                 else
9:                     entry ← (i, res) ↦ (o, res)
10:                end if
11:                DrSet_LC ← DrSet_LC ∪ {entry}
12:            end for
13:        end for
14:    end for
15:    return DrSet_LC
16: end function
```

Listing 1: DR Generation: Life Cycles

The set Σ_{AC} can be determined automatically in a pre-computation step by analyzing the security policy and mark every event occurring in the policy as access control relevant.

B. Static Access Control

For our DR algorithms, we will use the introduced task and process life cycles (Figure 3), as well as the process definition (Figure 1) as examples. Recall, a DR is a mapping of a trigger event to an access control relevant event

$$(\text{trigger event, 'resource'}) \mapsto (\text{ac relevant event, 'resource'}),$$

where the trigger event must precede the access control relevant event. Both events are bound to a respective resource, i. e., a process or a task. The overall generation of DRs is divided into three algorithms: GenDR-LC (Listing 1) is the core algorithm. It generates the DRs for a given life cycle (i. e., process or task life cycle) and corresponding resource. The second algorithm is GenDR-PD (Listing 2). As process definitions contain multiple tasks, this algorithm generates the DRs for each task of a process. The algorithm iterates over all tasks and calls the core algorithm on each of them to generate the relations. The third algorithm GenDR (Listing 3) is the one which orchestrates the overall generation of the relations. It calls the core algorithm GenDR-LC to generate the relations for the process life cycle and it calls GenDR-PD to generate the relations for all the tasks of a process definition. First, we introduce the core algorithm GenDR-LC for generating DRs from a given life cycle. The function GenDR-LC$(LC, res, (iEvent, iRes))$ takes three arguments: a life cycle model (*LC*), the resource for which the DRs are generated (*res*), and an event (*iEvent*, *iRes*) which helps to initialize the algorithm (on which we come back later).

Recall that the point when a pre-evaluation should be triggered is exactly that event (in a life cycle) which precedes access control relevant events. Along these lines the core algorithm (Listing 1) generates DRs by mapping each incoming event of a state to an outgoing access control relevant event. Consequently, we iterate over all states s of the life cycle *LC*. For each state we map the incoming events (i, res) to the outgoing access control relevant events (o, res) (lines 1.3–1.14) and store the outcome as DR $(i, res) \mapsto (o, res)$ (line 1.9).

The event which triggers the pre-computation for the first access control relevant events of a task (and its life cycle) lies in the event that happens before a task instance is created. There are two possibilities for such events. The first scenario is that the task is the first task of a process definition. In this case, the event which precedes the creation of the task instance is the creation of the process itself. Hence, the trigger for creating decisions for the first task of a process is the 'create' event for the process instance. We use the above mentioned third argument of the algorithm GenDR-LC and initialize the algorithm with the 'create'-event of the process (*'create'*$_{\text{PLC}}$, *'P'*). Giving an example for the generation of the DRs for the first task in a process (e.g., 'Task 1' of Figure 1) we call the algorithm as follows: GenDR-LC(*TLC*, 'Task 1', (create$_{\text{PLC}}$, 'P')). This results in the following set of DRs, where the initial event (*'create'*$_{\text{PLC}}$, *'P'*) maps on to the first two access control relevant events 'assign' and 'cancel' of the task life cycle in Figure 3. The other DRs are generated by further processing of all other states of the life cycle.

$$\begin{aligned}\mathit{DrSet}_{\text{Task 1}} = \big\{&(\text{create}_{\text{PLC}}, \text{'P'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 1'}),\\ &(\text{create}_{\text{PLC}}, \text{'P'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 1'}),\\ &(\text{assign}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 1'}),\\ &(\text{start}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 1'}),\\ &(\text{start}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 1'})\big\}\end{aligned}$$

The second scenario is that the task is within a process definition. In this case, the event which precedes the creation of the task instance is the creation of the task instance that precedes the current one. Hence, the trigger for creating decisions for a task within in a process definition is the 'create' event of its preceding task. We also use the above mentioned third argument of the algorithm GenDR-LC to initialize the algorithm with the 'create'-event of the current task (e. g., Task 2) its preceding task (e. g., Task 1): (*'create'*$_{\text{TLC}}$, *'Task 1'*). For illustration we call the algorithm as follows:

$$\text{GenDR-LC}(\mathit{TLC}, \text{'Task 2'}, (\text{create}_{\text{TLC}}, \text{'Task 1'})).$$

This results in the following set of DRs:

$$\begin{aligned}\mathit{DrSet}_{\text{Task 2}} = \big\{&(\text{create}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 2'}),\\ &(\text{create}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 2'})\\ &(\text{assign}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 2'}),\\ &(\text{start}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 2'}),\\ &(\text{start}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 2'})\big\}\end{aligned}$$

Finally, the algorithm GenDR-LC is also used to generate the DRs for a process life cycle, given a process 'P'. The function is called with the process life cycle (PLC), the resource 'P' as well as a default initializing event as arguments, i. e.,

$$\text{GenDR-LC}(\mathit{PLC}, \text{'P'}, (\text{create}_{\text{PLC}}, \text{'P'})).$$

This results in the following set of four DRs:

$$\mathit{DrSet}_{\text{P}} = \big\{(\text{create}_{\text{PLC}}, \text{'P'}) \mapsto (\text{cancel}_{\text{PLC}}, \text{'P'}),$$

```
1: function GenDR-PD(PD, PLC, TLC)
2:     DrSet_PD ← ∅
3:     for all t ∈ Q(PD) do
4:         for all i ∈ in[t] do
5:             if src[i] = q0(PD) then
6:                 events ← out[q0(PLC)]
7:                 rsc ← name[PD]
8:             else
9:                 events ← out[q0(src[i])]
10:                rsc ← name[src[i]]
11:            end if
12:            for all e ∈ events do
13:                DrSet_PD ← DrSet_PD ∪
14:                    GenDR-LCs(TLC, name[t], e, rsc)
15:            end for
16:        end for
17:    end for
18:    return DrSet_PD
19: end function
```

Listing 2: DR Generation: Process Definition

$$
\begin{aligned}
&(\text{start}_{\text{PLC}}, \text{'P'}) \mapsto (\text{stop}_{\text{PLC}}, \text{'P'}),\\
&(\text{stop}_{\text{PLC}}, \text{'P'}) \mapsto (\text{cancel}_{\text{PLC}}, \text{'P'}),\\
&(\text{start}_{\text{PLC}}, \text{'P'}) \mapsto (\text{cancel}_{\text{PLC}}, \text{'P'})\big\}
\end{aligned}
$$

Both the runtime and the output size of the function GenDR-LC depend on the size of the given life cycle (number of states) and its complexity (number of incoming and access control relevant outgoing events per state):

$$\big|\mathit{DrSet}_{\text{LC}}\big| \leq \sum_{s \in Q(LC)} \big|\mathit{in}[s]\big| \cdot \big|\mathit{out}[s] \cap \Sigma_{AC}(\mathit{LC})\big|$$

The algorithm in Listing 2 orchestrates the generation of DRs for all tasks of one process. Hence, it internally calls the function of Listing 1 for each task in the process and takes care that the initializing event (*iEvent*, *iRes*) is always set with the 'create' event of its preceding tasks. Calling the function GenDR-PD for our running example, i. e., GenDR-PD(PD, TLC, TLC) results in the following DRs:

$$
\begin{aligned}
\mathit{DrSet}_{\text{PD}} = \big\{&(\text{create}_{\text{PLC}}, \text{'P'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 1'}),\\
&(\text{create}_{\text{PLC}}, \text{'P'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 1'}),\\
&(\text{assign}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 1'}),\\
&(\text{assign}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 1'}),\\
&(\text{create}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 2'}),\\
&(\text{create}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 2'}),\\
&(\text{assign}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 2'}),\\
&(\text{assign}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 2'}),\\
&(\text{create}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 3'}),\\
&(\text{create}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 3'}),\\
&(\text{assign}_{\text{TLC}}, \text{'Task 3'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 3'}),\\
&(\text{assign}_{\text{TLC}}, \text{'Task 3'}) \mapsto (\text{cancel}_{\text{TLC}}, \text{'Task 3'})\big\}
\end{aligned}
$$

Both the runtime and the output size of the operation depend on the size of the given life cycles (number of states) and their complexity (number of incoming and access control relevant outgoing events per state) and the process definition.

```
1: function GenDR(PD, PLC, TLC)
2:     DrSet_CT ← ∅
3:     for all e ∈ out[q0(PLC)] do
4:         DrSet_CT ← DrSet_CT ∪
5:             GenDR-LCs(PLC, name[PD], e, name[PLC])
6:     end for
7:     DrSet_CT ← DrSet_CT ∪ GenDR-PD(PD, PLC, TLC)
8:     return DrSet_CT
9: end function
```

Listing 3: DR Generation

```
1: function GenerateRT(LC, res)
2:     RtSet ← ∅
3:     for all s ∈ F(LC) do
4:         for all i ∈ in[s] do
5:             RtSet ← RtSet ∪ {(i, res)}
6:         end for
7:     end for
8:     return RtSet
9: end function
```

Listing 4: Generating Revoke Triggers

In principle, for each incoming event of each task, the function GenDR-LC is executed. Thus, we can approximate the size of the generated set of DRs with:

$$\big|\mathit{DrSet}_{\text{PD}}\big| \leq \Bigg(\sum_{t \in Q(PD)} \big|\mathit{in}[t]\big|\Bigg) \cdot \big|\mathit{DrSet}_{\text{LC}}\big|$$

Finally, we join the three sets of DRs by calling the function GenDR(PD, PLC, TLC). It takes three arguments: the process definition *PD*, the process life cycle *PLC*, and the task life cycle *TLC*. This function (see Listing 3) calls GenDR-LCs and GenDR-PD.

C. Revoke Trigger Generation

In this section, we present a function to generate revoke triggers which remove those cache entries during runtime which are not needed any more (cf. Section III-E). Cache entries are revoked if either a task instance or process instance transitions into a final state. Hence, the revoke triggers contain the information on which events an instance transits into an end state. The algorithm GenerateRT (Listing 4) generates the respective set of revoke triggers for an LC and a given resource. For example, the process 'P' (Figure 1) and the presented life cycles (cf. Figure 3) result in the following set:

$$
\begin{aligned}
\mathit{RtSet} = \big\{&(\text{cancel, 'P'}), (\text{end, 'P'}), (\text{cancel, 'Task 1'}),\\
&(\text{end, 'Task 1'}), (\text{cancel, 'Task 2'}), (\text{end, 'Task 2'}),\\
&(\text{cancel, 'Task 3'}), (\text{end, 'Task 3'})\big\}
\end{aligned}
$$

Both the runtime and the output size of this operation depend on the number of transitions to an end state:

$$\big|\mathit{RtSet}\big| \leq \sum_{s \in F(PD)} \big|\mathit{in}[s]\big|$$

where $F(PD)$ denotes the set of end states of *PD*.

```
1: function GENDR-SOD(SP, PD)
2:     DrSet_SoD ← ∅
3:     constraints_SoD ← GETSODCONSTRAINTS(SP, PD)
4:     for all c ∈ constraints_SoD do
5:         for all currExclTask ∈ exclTasks[c] do
6:             for all e ∈ exclTasks[c] \ {currExclTask} do
7:                 DrSet_SoD ← DrSet_SoD ∪
8:                     {(Event[c], name[currExclTask]) ↦
9:                         (Event[c], name[e])}
10:            end for
11:        end for
12:    end for
13:    return DrSet_SoD
14: end function
```

Listing 5: DR Generation for SoD Constraints

D. Dynamic SoD Constraint

In this section, we describe an algorithm which generates the relations based on the two input parameters *SP* and *PD* where *SP* stands for *security policy* and *PD* is the process definition of the process for which the DRs based on SoD constraints should be generated.

The algorithm extracts the required SoD constraints from the security policy by calling the function GETSODCONSTRAINTS passing the policy and the process name as parameters (line 5.3). For each returned SoD constraint the algorithm generates the respective DRs such that as soon as one of the exclusive tasks occurs as a resource in an event new cache entries for the remaining exclusive tasks of the constraint are created. For the process model depicted in Figure 1 with SoD constraints, the following DRs are created:

$$\begin{aligned} DrSet_{\text{SoD}} = \big\{ &(\text{assign}_{\text{TLC}}, \text{'Task 1'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 2'}), \\ &(\text{assign}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 1'}), \\ &(\text{assign}_{\text{TLC}}, \text{'Task 2'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 3'}), \\ &(\text{assign}_{\text{TLC}}, \text{'Task 3'}) \mapsto (\text{assign}_{\text{TLC}}, \text{'Task 2'}) \big\} \end{aligned}$$

When Task 1 is assigned to a user, a new cache entry is created for Task 2. The new cache entry reflects that the user assigned to Task 1 may not be assigned to Task 2. The result also shows that a new cache entry for Task 1 is created after Task 2 is assigned. The latter relation shows that considering the order of executions in the workflow creates a cache entry for a task which is already finished at that point of execution. Due to the fact that the algorithm treats all SoD constraints in the policy equally regardless the order of the tasks, it creates more DRs than needed. Given the small number of SoD constraints usually defined for a process the additionally generated relations are a small overhead compared to the effort for analyzing the process definition according to the order of tasks, especially given advanced control flow constructs [19].

Additional revoke triggers have to be generated to prevent that during the generation of new cache entries invalid old entries may be accessible in the cache. Along these lines, a proactive caching implementation must guarantee that revoke triggers are handled with a higher priority than requests for cache entries. Due to space limitations and its similarity to previously presented algorithms we skip the algorithm for SoD related revoke triggers.

	without caching	with caching	pre-evaluation	speedup
model 1	769 ms	6 ms	1964 ms	128
model 2	1267 ms	8 ms	2717 ms	158
model 3	2687 ms	43 ms	5146 ms	62

TABLE I
PROCESSING TIMES FOR ACCESS CONTROL CHECKS AT RUNTIME

V. CASE STUDIES

We implemented the proactive caching framework in Java including the algorithms introduced in this work [8]. For evaluation, we generated caching heuristics for three different types of workflow models and simulated them on a Pentium 4, dual core machine (2.0 GHz) with 1.5 GB of RAM.

First, we consider a linear workflow having only one available path of execution (i. e., no loops and no splits). Second, we consider a model with a small branching factor which influences the workflow execution with a greater number of DRs and, hence, a greater number of pre-evaluated access decisions. The third model contains multiple branches and loops, which also increases the overhead for generating pre-evaluated access decisions as more possibilities for the process flow must be considered.

The goal of the simulation is to compare the effort required for performing access control evaluations in an environment where no caching is applied with an environment where our proactive approach is implemented. We simulated the execution of all three workflow models. In each case an instance of the workflow was created and all its tasks successively executed. Access control requests during execution where made against an access control policy with 1500 users (distributed on 15 roles), and a set of 1700 permissions.

Table I summarizes the results of our evaluation. For each model, we report the accumulated processing times (in [ms]; without caching and with proactive caching) required to process all access control requests to be evaluated during its execution. Moreover, we report on the time needed for computing the cache entries (pre-evaluation). Our evaluations show a performance gain of at least a factor 60, whereas the overhead time required to generate the pre-evaluated access decisions is only twice as much compared to the scenarios where no cache is used. Our evaluations show that the relation between overhead and the regular time decreases with increasing complexity of the workflow model.

VI. CONCLUSION

A. Related Work

Improving the performance of access control decisions in a distributed environment is a widespread research topic. Whereas we focus on performance gains by using high-level process models for adapting a generic caching architecture many research approaches are focusing on optimizing the PDP itself [10] or caching PDP results in the PEP [1], [2], [7], [16]. All these cache architectures share the same two limitations

compared to our approach: first, they do not anticipate cache queries, and second they do not approximate state-dependent parts (context information). Especially the second point is, in our opinion, increasingly important as modern business regulations require dynamic concepts, like SoD, that cannot be cached easily. Thus, caching solutions that require a static policies and a stateless PDP are suffering from a very high miss rate in such dynamic scenarios we are looking at. There are different approaches for optimizing the security polices with respect of the evaluation complexity [10], [12].

The most closely related work is the secondary and approximate authorization model (SAAM) [4] and its application to systems using RBAC [20]. The core idea of SAAM is the introduction of a secondary decision point (SDP). A SDP is in principle caching approximate results of the PDP. The concrete form of approximation depends on the requirements of the SDP (e. g., with respect to its safeness and correctness) and the underlying authorization model (e. g., RBAC or Bell-laPadula). Overall, this approximations increases the hit rate of the caching solution substantially. With this work, we share on the one hand the experience that caching exact results of the PDP leads to many cache misses and on the other hand the vision of using abstract models for adapting the runtime behavior [5] of the caching infrastructure.

Whereas SAAM focuses on advanced heuristics for approximating PDP results, we decided to approximate the PDP results by strictly separating the static access control model from the dynamic context information. Hence, our work focuses on using abstract models for adapting the caching architecture for an efficient pre-computation of cache entries.

B. Conclusion and Future Work

We propose ProActive Caching which is a caching strategy tailored to business process-driven environments. It provides caching of access control decisions based on high-level system descriptions which are used to automatically generate workflow-specific caching heuristics. These heuristics enable a proactive caching framework to anticipate future access queries at runtime, pre-evaluate the respective access requests in advance, and store the results as cached access decisions such that they are available when needed.

In this paper we presented algorithms for the automatic generation of workflow-specific caching heuristics based on input sources such as a system's process and task life cycles, process definitions, as well as SoD security constraints.

We analyzed our approach by generating caching heuristics for three types of workflow models. The results show that the caching approach is very effective. The performance for access control checks during runtime increases at least by a factor of 60 if the cache is used, whereas the effort to be put into the system remains by only twice as much processing time compared to a system where no caching is implemented.

For future work, extending our framework for security policies that require fine-grained delegation concepts is an interesting line of research. Along these lines, the transfer of the proactive caching strategy to new applications domains is a further aspect to be investigated. For example, our technique could be used for pre-checking the availability of resources needed for the execution of a given task.

VII. Acknowledgments

We thank Robert Fies for valuable discussions on the topic of this paper. This work has been supported by the German "Federal Ministry of Education and Research" in the context of the project "SoKNOS."

References

[1] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 81–95. IEEE Computer Society, 2005.

[2] K. Borders, X. Zhao, and A. Prakash. CPOL: high-performance policy evaluation. In V. Atluri, C. Meadows, and A. Juels, editors, *ACM Conference on Computer and Communications Security*, pages 147–157. ACM Press, 2005.

[3] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[4] J. Crampton, W. Leung, and K. Beznosov. The secondary and approximate authorization model and its application to Bell-LaPadula policies. In *Proceedings of the ACM symposium on Access control models and technologies*, pages 111–120. ACM Press, 2006.

[5] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, 2006.

[6] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman, Inc., 1979.

[7] G. Karjoth. Access control with IBM Tivoli access manager. *ACM Transactions on Information and System Security*, 6(2):232–257, 2003.

[8] M. Kohler and R. Fies. ProActive caching – a framework for performance optimized access control evaluations. In *IEEE International Symposium on Policies for Distributed Systems and Networks*. IEEE Computer Society, 2009.

[9] M. Kohler and A. Schaad. ProActive access control for business process-driven environments. In *Annual Computer Security Applications Conference*. IEEE Computer Society, 2008.

[10] A. X. Liu, F. Chen, J. Hwang, and T. Xie. Xengine: a fast and scalable XACML policy evaluation engine. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 265–276. ACM Press, 2008.

[11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42. USENIX Association, 2001.

[12] P. Miseldine. Automated XACML policy reconfiguration for evaluation optimisation. In B. de Win, S.-W. Lee, and M. Monga, editors, *SESS*, pages 1–8. ACM Press, 2008.

[13] J. Nielsen. *Usability Engineering*. Academic Press, 1993.

[14] R. S. Sandhu, D. F. Ferraiolo, and D. R. Kuhn. The NIST model for role-based access control: towards a unified standard. In *ACM Workshop on Role-Based Access Control*, pages 47–63. ACM Press, 2000.

[15] W. Shin and H. K. Kim. A simple implementation and performance evaluation extended-role based access control. In *Proceedings of the 4th WSEAS International Conference on Software Engineering, Parallel & Distributed Systems*, pages 1–5. World Scientific and Engineering Academy and Society (WSEAS), 2005.

[16] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *Proc. 8th USENIX Security Symposium*, Aug. 1999.

[17] The JBoss Group. jBPM. http://www.jboss.com/products/jbpm, 2008.

[18] W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[19] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[20] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in RBAC systems. In *Proceedings of the ACM symposium on Access control models and technologies*, pages 63–72. ACM Press, 2008.