ProActive Access Control for Business Process-driven Environments*

Mathias Kohler and Andreas Schaad
SAP Research
mathias.kohler@sap.com
andreas.schaad@sap.com

Abstract

Users expect that systems react instantly. This is specifically the case for user-centric workflows in business process-driven environments. In today's enterprise systems most actions executed by a user have to be checked against the system's access control policy and require a call to the access control component. Hence, improving the performance of access control decisions will improve the overall performance experienced by the end user significantly. In this paper we propose a caching strategy which pre-computes caching entries by exploiting the fact that the executions of business processes are based on the execution of actions in a predefined order. We propose an accompanying architecture and present the results of our conducted benchmark.

1. Introduction

Short response times for user interactions are of elementary relevance. Users expect that a system reacts instantly on their input. System response times up to 0.1 seconds of delay are perceived as direct interactions with the system. Times above this value are already recognized by the user as interruption [3]. In business process-driven environments fast response times are required when a user wants to claim a new task, when she wants to perform the task she claimed, or whenever she wants to execute any process management interactions within the workflow system. As important as it is to achieve optimized response times for the overall system, we equally have to consider design and implementation of each single component which may impact performance with respect to the complete system. In today's large enterprise systems, most actions executed by a user have to be checked against the system's access control policy. Thus, in today's systems most calls to basic functions require a call to the access control component. Therefore, improving the performance of evaluating access control decisions will significantly improve the overall performance experienced by the end user.

One solution for improving the performance of the access control evaluation process is caching. A naïve caching approach for role-based systems is to store access control decision responses and reuse them whenever an identical access decision request occurs. The caching strategy presented in this paper exploits the fact that the executions of business processes are based on the execution of actions in a predefined order. We take advantage of this fact and evaluate relevant access decision responses upfront for immediate subsequent actions of a currently executed process. The advantage is that the cache with pre-computed entries contains exactly those entries most likely to be needed next.

Our approach is to define dependency relations that specify the relation between events of executed actions in a business process system and the access decisions which can be pre-computed given such an event. These dependency relations are the basis for pre-evaluating exactly those access responses needed during the ongoing execution of the process. The cache will contain exactly those entries which will potentially be needed next, such that we expect a high cache hit rate and hence a great potential to decrease the response time compared to regular access decision evaluations.

State of the art caching techniques for access control solutions are only supporting static policies, i.e., policies that do not depend on context information. Especially in the area of business process executions, dynamic constraint types [4] such as separation of duty or cardinality constraints come into play. Such constraint types rely on dynamic context information (such as the user's history of performed tasks, or the current time).

Our approach to cache decision responses considers the before mentioned constraint types which rely on
dynamic context information. We provide a caching strategy which enables their caching.

We implemented and benchmarked our proposed caching solution to analyze the effect of our caching strategy as opposed to direct evaluation. The results show a significant performance gain on the caching side.

In this paper our contribution is threefold: First, we present a caching strategy for caching policy decisions that supports dynamic aspects of access control (e.g., separation of duty).

Second, we present a heuristic for pre-computing cache entries based on abstract (business) process models which, for example, allow improving the overall response time for user centric processes.

Third, we show the results of our benchmark tests which state a significant performance increase for the use of the proposed caching strategy.

The remainder of this paper is structured as follows. In the following section we will give background information on business processes. In section 3 our proposed caching strategy is introduced, where section 4 depicts the accompanying architecture. In section 5 we present the results of our conducted benchmark. A critical evaluation of our solution is given in Section 6. Section 7 concludes the paper.

2. Background

Business processes "focus upon the production of particular products" [5] realized by a sequence of tasks. Processes may be modeled in total or in parts as workflows.

Business process executions may be fully automated, partially automated, or completely user centric. Human based executions comprise the process is user centric and, hence, relying on user interactions to perform the single tasks. The workflow system's responsibility is the process and task management, but a human person is the one eventually claiming and executing the tasks for which she is responsible. We concentrate on user centric workflow executions.

Business process execution happens in three Execution Layers: in the user interface layer, the business process layer, and the business object layer (see Figure 1).

The user interface layer provides the interface for a user to access the work lists as well as the application interfaces which enable the user to perform a task. The business process layer is responsible for the process and task management which especially deals with the state control of processes and their tasks.

![Figure 1: Business Process Execution Layers](image)

The business object layer provides the access to methods of business objects or externally located web services. Business objects provide the basic functionality of the system on which the workflow and especially the task execution is built on. Typically provided functionality are methods for querying and modifying data located back-end systems.

2.1 Process Layer

We present process and task life cycles on basis of the models presented in [6], mainly concentrating on those stages relevant for our caching strategy.

![Figure 2: Process Life Cycle](image)

Whenever a process is to be executed, an independent process instance is created based on a selected process definition. The states a process instance can adopt are Initiated, Running, Completed, and Terminated. Instantiated is the state after the process instance was created. Running reflects that at least one instance for the first task of the process has been created and assigned to a user. The states completed and terminated express equally that the execution for this process instance has been finished. The difference is that a completed instance reflects all
task instances have been completed; a terminated instance reflects the process execution was intentionally canceled outside of the ordinary process execution. Figure 2, gives an overview of the mentioned states.

Every state of the process life cycle can be reached through transitions. They transfer the process instance from one state to another. We distinguish between two types of transitions: explicit and implicit. The executions of explicit transitions are triggered by user interactions. They include createProcess, restartProcess, and cancelProcess. CreateProcess leads to the state Initiated and is called when a user starts a new process execution. A user may also call restartProcess or cancelProcess which brings the process instance to the states Initiated or Terminated respectively. The executions of implicit transitions include startProcess and completeProcess. StartProcess sets the workflow engine into the state Running. It is implicitly called when the first task instance of a process is created and assigned to a user. CompleteProcess is implicitly called as soon as all task instances for the current process instance are finished.

Obviously, if users are involved to invoke state transitions, there is at least the question whether these users are allowed to initiate the state change. Each of the user driven transitions described above is required to be controlled by an access control mechanism. CreateProcess should only be called by users which have the permission to start it. A purchase order may, for instance, only be started by users from the sales department. CancelProcess or restartProcess may, for instance, only be initiated by administrators or the users who originally started the process.

Similar constraints also apply for the task life cycle which we will show next.

2.1.2 Task Life Cycle

![Figure 3: Task Life Cycle](image)

The task life cycle comprises the states Active/Not Assigned, Active/Assigned, and Completed. This life cycle corresponds to user centric task execution. For automated tasks, obviously the differentiation between assigned and not assigned can be neglected. A task instance is immediately in state Active/Not Assigned when created. If a user decides to claim the task, the transition claimTask is called and the user is assigned to the task. An access control check verifies the security policy that allows the action accordingly.

A user can withdraw from a task by calling the method revokeTask; the task's state returns to the state Active/Not Assigned. In cases where revoking a task is not allowed, an access control check prevents the user from revoking the task. When the task is finished, endTask is called. An access control check at this stage checks that, for instance, a preset processing time is not exceeded.

2.2 User Interface Layer

The user interface layer provides two types of interfaces for process execution. One is for process management, the other serves as input mask for single task executions.

The interface for process management comprises the possibilities to start a new process, cancel or restart a running process as well as to handle task instances. Before a user can start working on a task she must claim the task. All tasks available to be claimed are listed in a General Worklist (GWL). The user opens the GWL and claims the task(s) she wants to perform next.

Of course, the GWL should only display those tasks a user is actually allowed to claim (which are usually only a fraction of the total number of tasks active in a system). For this reason, for each task instance created in the system a potential list of owners is generated. This list is created based the roles a user must possess to perform the task (RBAC, [13]). Fact is with only considering roles (rather than also other types of constraints, e.g., SoD) the user might get displayed tasks which she still cannot claim. Hence, whenever the GWL is displayed a second access control query checks for every task instance in the list whether it could be claimed by the user. Especially for such "bulk-access-control-checks" we expect a significant potential of performance benefit wrt the experienced response time for the user.

2.3 Business Object Layer

The business object layer is the operating unit performing the executions requested by a user, where all data modifying queries or data selection requests are performed. This layer does not keep track of which of the calls are related to which process or task instance and which are independent. All calls are treated stateless. Still, access control is needed on this layer as in today's web service-based enterprise systems the objects are accessible independently of the above layer. To guarantee that a specific call is only executable if the user is actually assigned to a related
task instance, every method call on the business object layer is secured by an access control check. Taken the example from the previous section, the user whose task it is to select or create a new supplier, the back end system must guarantee that (only) the user performing the current task is eligible to query the list of suppliers or create a new supplier to continue the process.

Further access control needs on this layer are typically that a user must hold a specific role, or that parameters transferred for the business object call may only be within a certain range (e.g., orders are only accepted up to an amount of 100k EUR).

### 2.4 Access Control Reference Model

Although the presented three presented layers may be tightly linked with each other from an access control perspective, it is still the case that each layer can be accessed independently. This means methods of business objects can also be called without the necessity that the call has to be mediated by a workflow engine; the functionality exposed by the workflow engine can also be used by third party software delivering their own user interface adapted for their needs. For this reason it is crucial that each of the presented layers incorporates its own enforcement components to realize access control enforcement on each layer.

In large enterprise systems access control is usually based on the request-response paradigm. It comprises 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. Each of the above presented layers contains one or more PEPs to enforce the decisions accordingly.

### 3 Pre-emptive Caching Strategy

#### 3.1 Related Work

Response times for user interactions are of elementary relevance [7]. Responses exceeding 0.1 seconds are already perceived as interruption [3]. Delays beyond 1 second are only acceptable if in the execution of an application a context change happens.

The decision evaluation process can be improved wrt performance using different strategies. One is the specific optimization of the policy evaluation process. The optimization in this case is tailored to a particular policy structure. An example is given in [8] which is based on the XACML policy structure. The author improved the speed of the actual policy evaluation by refactoring the policy such that a performance increase of the overall policy evaluation can be achieved. The authors of [11] optimized the evaluation according to the location where the policy is stored.

Another strategy to optimize access control performance is in returning a cached access control decision response instead of performing the actual access control decision evaluation.

In [1] and [2] a caching architecture is presented which claims to increase the probability of cache hits by generating responses by inferring new decisions from already cached responses. The presented caching strategy is limited on specific types of policies. The caching strategy of [1], for instance, relies on systems which solely use RBAC.

In [12] the authors propose the idea of speculatively pre-computing and distributing "junk" authorizations and plan to develop methods for efficient predictions of which authorization request(s) should be computed next, based on the history of prior requests.

The strategy presented in this paper is specifically tailored to business process-driven environments. It provides a method to pre-evaluate exactly those cache entries which will most potentially be needed next. Here, we do not rely on previous access requests received, but use the explicitly available knowledge of the order in which processes and therefore access control relevant actions will be performed.

### 3.2 Caching in Business Process-driven Environments

The goal of the caching strategy presented in this paper is to decrease the overall response time experienced by the user when she is interacting with the system. In today's large enterprise systems, most actions executed by a user have to be checked against the system's access control policy. To reach the overall goal we concentrate on performance gains we can make by reducing the response time attributed to access control evaluations. The response time can be reduced if the most costly steps during an access control evaluation can be shortened or totally removed.

Caching of access control responses is one way of reducing response times as the cost for the evaluation procedure can be saved - if a cache entry for the access request is available.

#### 3.2.1 Overview of Caching Strategy

The objective of the caching strategy presented in this paper is to provide a solution that optimizes the availability of cache entries (which in turn is the vehicle to increase the overall experienced performance). Our solution aims at confirming the assumption that a cache is most effective if it only
contains exactly those entries currently needed (and, hence, only for exactly that period of time when they are expected to be needed). In business process environments, there are three main sources from where we can obtain the information to predetermine the actions which will be performed in immediate future.

The first source comprises the specifications of a system's process and task management. As seen in the previous sections, if a process or task instance is in one particular state, only a small subset of transitions determines which subsequent actions are possible (to reach the next state).

The second source to determine upcoming actions is the link between the task of a workflow and the business object (BO) of the back-end system. As already described, the BO is called to perform data selection and modification operations initiated from the workflow layer. This means, as soon as a task instance is created, it is very likely that the BO's exposed methods relevant for the task execution will be called. This may happen during the period of time when the task instance is active.

The third source is the process definition, referring to the control flow perspective. The process definition specifies the order in which the tasks of a process are executed. It is clear that, for instance, if task 1 and task 2 are in subsequent order the execution of task 2 will start as soon as task 1 is completed.

Our approach is to use this knowledge and prepare the cache in such a way that access requests can be directly answered by already pre-calculated and cached decision responses. The general caching strategy for business processes we propose is as follows. The cache contains a set of cache entries. Each entry is a pre-fetched decision response for an action which is expected to be called as one of the upcoming actions performed during the execution of a business process instance. In fact, an entry is exactly generated for one specific action, resource, subject, and process instance which together form what we call target. Further, each cache entry contains a result element which is the pre-evaluated result for a given target. If an actual decision request maps to the target, the cache entries’ response element is returned as the decision response.

The cache entries are built upon the occurrence of predefined events. We call them trigger events (TE). A trigger event is, for instance, the execution of a specific action of the process life cycle (e.g., createProcess).

The TE provokes the creation of a cache entry based on a predefined successor request (SR). The SR is the access request expected to happen after the TE occurred. An example is the action cancelProcess which is an action potentially called after the process instance was created. At some point in time the process ends, either by executing the cancelProcess method or simply because all tasks have been finished. This is the time where the cache entries (generated specifically for the actually running process instance) have to be revoked from the cache as they are of no further use. A second trigger, called revoke trigger (RT) defines the action(s) on which the removal of the cache entry is performed. The definition of the trigger event, the successor request, and the revoke trigger are summarized within a tuple of three elements called dependency relation (DR) = \{TE, SR, RT\}. We will elaborate on this concept in section 3.2.4.

3.2.2 Constraint Types and Context Information

Access control for business processes does not only comprise role-based permissions but also other constraint types such as separation of duty (a user may only perform one of two exclusive tasks), binding of duty (a user performed one of two specific tasks must also perform the second one), cardinality constraints (a user may only perform a predefined number of iterations of a task in the same process instance), attribute-based constraints (a certain value of a business object must respect a predefined condition), or time and date-based constraints (the execution of an action is dependent on date and time constraints). Examples are given next.

Role-based permission: In user centric workflows the basic building block for every action during a process execution is that the required permission is related to a role. A user must possess a certain role which provides her with the permission to perform the respective action on a chosen resource. An example would be that a task in the process for private customer handling may only be performed by users that are members of the role PrivateCustomerClerk.

Time or Date-based constraints: Time and/or Date-based constraints restrict a permission for the execution of an action based on time or date conditions. A basic example is that a task may only be performed during working hours and therefore the condition 6 a.m. \(\leq\) current time \(\leq\) 8 p.m. must be evaluated to true. Current time in this case is a dynamic context information which is directly fetched from a context provider whenever the condition has to be evaluated. Another example is that the duration of a task execution is restricted such that a user must complete a task within 30 minutes. Obviously, also in this case a context provider (e.g., located with the workflow engine) keeps track of the progressed time and reports the elapsed time when needed for the evaluation of the condition.

Dynamic Separation of Duty constraints: Constraints based on dynamic separation of duties (SoD) define that certain tasks of a business process are exclusive and may not be performed by the same person. This means as soon as a user claimed one of
the exclusive tasks, the user may not claim any of the other exclusive tasks.

**Binding of Duty constraints:** The binding of duty constraint (BoD) between two tasks defines that a user performing one of the two tasks must also perform the second one.

**Attribute-based constraints:** There are also constraints based on context information which is available in the system but do not fit any of the previously described constraint types. The name attribute-based refers to information about a subject or object.

### 3.2.3 Cache Entries

A cache entry must hold the information needed to: (1) obtain the relevant cache entry from a set of entries for a decision request; (2) provide the corresponding decision response (PERMIT or DENY); and (3) maintain a reference to the corresponding process instance such that the entry can be related to a specific process execution.

The cache entry CE(S, A, R, P, [OC], PIID) is a tuple of 6 elements which are described as follows.

- **S:** S is the subject identifier for which the cache entry is generated.
- **A:** A is the respective action identifier.
- **R:** R is the respective resource identifier.
- **PIID:** PIID stands for process instance ID and is part of the cache entry to link a cache entry to a specific process execution instance. (This is necessary as there might be several instances of the same process running in parallel.)
- **S, A, R and PIID** comprise the target of the cache entry. The target is needed to select the respective entry for a certain access request.
- **OC:** OC is a set of open constraints. Open constraints are Boolean expressions which have to be evaluated before a cache entry's permission can be used as response. We will elaborate on OCs in section 3.2.5.
- **P:** P is the permission which is returned as a decision response if the cache entry maps to a decision request. P can hold the responses PERMIT or DENY. P assumes that all open constraints evaluate to true. Only if this is the case then P is returned as decision response.

### 3.2.4 Dependency Relations

We already introduced the notion of a dependency relation (DR) in the previous section 3.2.1. In this section we will elaborate on its definition and purpose.

The DR comprises three elements, namely the definition of a trigger event (TE), a successor request (SR), and a set of of revoke events (RE). We write it as DR(TE, SR, {RT}). We will describe each of the elements below. For a better understanding, however, we introduce the notion of an event first.

During the execution of a process events happen. An event is the transition from one state to another state given a process or task instance. An event(action, resource, subject, piid, [tiid]) is a tuple of five fields (whereas the field tiid is optional; tiid stands for task instance ID and uniquely identifies the respective task instance if available). Each of the fields can adopt values describing the occurring transition during the process execution. A typical example is given as follows. Assume a user with the subject ID 'Alice' creates a new process instance of the process 'PurchaseOrder'. The access control check approves that Alice has the permissions to create the process and eventually a process instance with a process instance ID (piid) 4711 is created. The event in this case is event(createProcess, PurchaseOrder, Alice, 4711).

The purpose of a dependency relation is to define on which events occurring in the system new cache entries should be created and on which events in the system obsolete cache entries should be revoked. In the following paragraph we introduce the three elements TE, SR, and RE of a dependency relation.

The **trigger event** TE(action, resource, subject) describes the event on which a new cache entry is created. It defines an action, resource, and subject element. Events of the system can match with trigger events if the elements of the event are equal to the elements of the trigger event. If this is the case, a new cache entry is created. The elements action, resource, and subject are specified as values (e.g., action = 'createProcess'). Alternatively, they can be defined as random. The random placeholder helps specifying dependency relations which do not only match to events with specific action-resource-subject-combinations, but also being relevant for an event where, for instance, only the action and resource elements are relevant but the subject element may arbitrary. An example for a trigger event is TE('createProcess', *, *). It states that the TE will match with any event having the action 'createProcess', independent of the resource and subject.

The **successor request** SR(action, resource, subject, piid) defines the action, resource, subject and process instance ID for which a new cache entry is generated. An example stating that a cache should be created for the action 'cancelProcess', the resource 'PurchaseOrder' and the subject 'Alice' would be specified as SR('cancelProcess', 'PurchaseOrder', 'Alice', event.piid). The process instanceID is taken from the event that triggered the cache generation.
We can also specify a cache entry which should be created for the same resource as given with the event triggering the cache creation; we define a reference to the resource of the event as resource element:

SR('cancelProcess', event.resource, 'Alice', event.piid).

There are situations where given an event multiple cache entries should be created; one cache entry for each subject out of a set of subjects. This requires that the field subject in SR must possibly hold the set of subjects or refer to an external function to be called when the required set of subjects should be determined.

The third element of a dependency relation is a set of revoke events, RE(action, resource, subject, piid). These elements are responsible that upon the occurrence of a certain event the cache entries generated are revoked from the cache. Note that each dependency relation defines the events which revoke exactly those cache entries that have been provoked by the dependency relation to be created.

An example for a revoke event definition is RE('endProcess', 'PurchaseOrder', 'Alice', event.piid). It states that the cache entry previously generated should be removed whenever the event('endProcess', 'PurchaseOrder', 'Alice', event.piid) occurs. During runtime, event.piid has to be replaced with the process instance ID of the initial event which triggered the cache entry generation. In the following we give a few examples for dependency relations.

The first example defines a dependency relation with an event trigger for the action 'createProcess' and no further specification of subject or resource. Further, the dependency relation defines a successor request which generates cache entries for the action 'cancelProcess', for the same resource which initially triggered the event, and for each user which is according to the role resolution a potential user to perform such an action. To retrieve the list of potential users we use the informal function call pdp.getPotentialUserList(action, resource) which retrieves the list of potential users from the PDP. The last element of the dependency relation states a set of REs that defines that events with the action 'endProcess' or 'cancelProcess' will revoke the cache entries previously generated. The referencing link between the cache entries and the revoke events is the process instance id.


The second example defines the dependency relation between two consecutive tasks 'CheckCustomerRating' and 'ApproveCredit' of a process 'LoanOrigination'. It states if the first of the two tasks is completed, cache entries for the upcoming second task can be generated. Also in this case, we generate a cache entry for each user which potentially may claim the task. These cache entries are used when a user eventually wants to claim the task instance for 'ApproveCredit'. Dependency relations similar to this example can be defined for the complete process such that there is one dependency relation for all consecutive tasks within a process.


The third example demonstrates a dependency relation for a task where the execution is conditioned on the current time. The trigger event for the generation of the cache entry is the action 'claimTask' where the user is assigned to the task. The successor request shows the action, resource, and subject which is used to generate the cache entry. Further, there is the open constraint 'wfe.Now( ) ≥ 6.00 AND wfe.Now( ) ≤ 17.00' stating that the task may only be completed during working hours. Again, we use the informal notation wfe.Now( ) to retrieve the needed context information from the workflow engine (wfe), the current time in this case.


The example also specifies revoke events, especially for the events when a user revokes or ends a task. In both cases the cache entry should be removed from the cache. Obviously, it should also be removed if the process is cancelled.

As demonstrated, one of the sources for DRs are the transitions of the life cycles introduced in section 2.1. They represent a very common view on possible life cycles; still every vendor implements its own versions of them which might differ slightly. Hence, dependency relations based on such task and process life cycles have to be defined accordingly.

3.2.5 Cache Entry Generation

The cache entry will be generated wrt the dependency relation (DR) (see above).
We expect the majority of access control decision requests during a process execution require RBAC evaluation which does not depend on dynamic context information. For the creation of a cache entry in such cases, the subject for which the cache entry should be created, the respective action and resource is prepared as an access decision request and sent to the PDP for evaluation. We call this a fake request. The response of the fake request is used as cache entry.

The evaluation of such a fake request may also require the evaluation of other than role-based constraint types (cf. section 3.2.2) requiring time, date, and attribute-based context information which change or may change during a process execution without further notice. We classify such context information which underlies steady or unexpected changes as dynamic context information. The challenge is the fact that caching decisions which are based on such information possibly become invalid due to context changes.

We provide a solution which makes caching of decisions possible - even if the evaluation requires dynamic context information. To be able to cache such access decisions we introduced an additional element for a cache entry: open constraints (cf. section 3.2.3).

Security policies are usually structured in such a way that permissions which are given to a role are further restricted by conditions: a user might be member of a role that has the permission to execute Task A; an additional condition, however, restricts access to it only within working hours. Such conditions, being time or date-based or any of the other in section 3.2.2 presented constraint types, are usually expressed such that they have to evaluate to true that the policy evaluates to a specific effect (i.e., PERMIT or DENY). For example, the condition 6 a.m. ≤ Now( ) AND Now( ) ≤ 8 p.m. must evaluate to true such that access is granted for Task A. Conditions requiring such dynamic context information for their evaluation are called open constraints (OC).

A cache entry is a stored response from a fake request: it contains the final decision of a PDP’s request evaluation. The result of a cache entry assumes that all open constraints evaluate to true (no matter of the values the actual dynamic context information during evaluation holds). Hence, given the above example, the permission P stored as response in the cache entry assumes that the current time lies between 6 a.m. and 8 p.m. independent of the actual time during the evaluation.

If the cache entry is eventually used to answer an access request, then the open constraint is checked against the real values of the dynamic context given the current status of the system. If all OCs evaluate to true, the stored permission holds (as during the permission's evaluation as they were equally assumed to evaluate to true) and P is returned. If one of the OCs evaluates to false, the cache entry can not be used and a regular evaluation with the PDP must be done.

4. Caching Architecture

The caching architecture extends the above mentioned access control architecture with four additional components, namely: an access decision cache, a cache management component, an event listening component, and a dependency storage. The cache stores all created cache entries for policy decisions. Whenever a decision request is made which matches a cache request, the cached response is returned.

For the evaluation of open constraints the cache might have to call the respective context providers for context gathering. The cache can either be located on the application side functioning as proxy for each access decision request, or it can be located at the server side where the policy decision point is located as gathering point for all requests.

Figure 4: Caching Architecture

We suggest that the policy decision point will be used to evaluate fake requests and return the decision which will be used within the cache entry. As described above, we have different types of constraints. For our cache creation, role-based permissions as well as SoD, BoD, and cardinality constraints can be evaluated as for any regular decision request as well. Constraints of type date, time, and attribute-based must considered true for the evaluation of the fake request. Hence, if the PDP recognizes a request from the cache management component it needs the additional functionality to react accordingly.

Furthermore, the cache needs the functionality to evaluate open constraints. In Figure 4 we depicted this by applying an Open Constraint Evaluator to the Cache.
5. Benchmarking the Architecture

Regarding the actual efficiency of our suggested approach the following criteria now require evaluation:

- Comparison of decision request evaluation performance regarding the general absence or presence of our proposed caching component.
- Comparison of proposed caching strategy wrt to different constraint types

We implemented the proposed architecture and benchmarked the two mentioned goals to analyze the proposed strategy.

Our implementation includes a PDP component for which we used a rule engine. In particular, we used JBoss DROOLS [10] with which we developed a declarative approach to implement an access decision evaluation engine. For the benchmark we generated role-based permissions, policies with SoD constraints, as well as policies with attribute-based constraints. Our test policy contains 1500 users, assigned to 15 roles (100 users each), and an overall number of 1700 permissions.

This was tested against a filled cache. The amount of entries is based on the assumption of 1000 parallel running process instances; every instance has approximately 60 cache entries (10 pre-computed entries for the actions cancelProcess and restartProcess, 40 entries for concurrent active task instances, and 10 pre-computed entries for BO Calls). We assumed according to our strategy, that for each access request, a cache entry is available if the cache is used.

The context information retrieval needed for the access evaluation (e.g., the list of executed tasks by a user) is realized by querying a local database. The PEP Cache is implemented as hash table where the target of a decision response functions as key. The values stored in the table are the pre-computed permission (i.e., PERMIT or DENY) and optionally the open constraints.

The tests were performed with JAPEX [9], a micro-benchmarking framework run on an Intel Pentium Dual Core system with 2.0 GHz and 1.5 GB RAM.

The results are as follows. We conducted three test cases where each of them has been performed with and without the cache being used. All test cases performed 50 access control request in row against the PDP. This reflects the scenario if a user enters her worklist and all active tasks for which she is potentially responsible are checked whether she is currently allowed to perform them. On the PDP’s side the policies are implemented. Test Case 1 uses 50 requests which on the PDP’s side purely needs role-based evaluation. This means, the response times shown in the first column of Figure 5 are based on RBAC evaluations. The second test case uses a mixture of requests. 80% RBAC of them were purely evaluated using RBAC, for the remaining 20% SoD evaluation was necessary (incl. the history selection from the database). The results are shown in the second column of Figure 5. The third test case uses a mixture of requests where 60% of the requests were evaluated based on RBAC policies, 20% based on SoD policies and the remaining 20% based on attribute-based policy evaluation.

The results show the overall response times for 50 access requests in row. The more external context data is required (e.g., history data of a user) the longer takes the response. This also holds for cache entries. Those with open constraints (i.e., attribute-based constraints) require additional context retrieval (cf. Section 3.2.5); the context request clearly increases the response time (see case three). Obvious is the significant overall difference between response times with and without cache. In this implementation no remote and distributed communication was simulated.

6. Concept Evaluation

The presented concept relies on the fact that business process executions have predefined states and every state has a limited set of actions which might be performed next. This also means the developed dependency relations for a particular system rely on the fact that the given information about process and task management as well as process definitions is valid. As this most certainly remains true for the process and task management, process definitions may change. It is clear that in such a case the dependency relations have to be adopted. The same holds for changes in the policy definition, especially regarding, for instance, newly added or changed open constraints. Also, if the
policy changes the cache entries have to be updated. Typical approaches are to flush the cache whenever the policy changes.

We propose the pre-evaluation of access decisions only for subsequent upcoming events, rather than pre-computing entries for the complete process. The reason is twofold. First, making a complete pre-evaluation would pollute the cache with entries not needed at the current point in time as most cache entries would be created for tasks which are not active, yet. This would increase the response time for the cache without further benefit. Second, some cache entries which, for instance, are based on the evaluation of SoD constraints, need a user's history information to compute a respective result. Obviously, this type of information is only available if the user already claimed or completed the related tasks. Hence, the assignment of a user to a task (which also is part to the SoD constraint) is the earliest event where such types of entries can be reliably generated.

However, there is still flexibility when cache entries can be pre-computed. The period of time lies between the occurrence of the trigger event and the event for which the cache entry is generated. Therefore, within this boundary entries can be generated whenever the system responsible for creating cache entries has spare-time.

Still open for a complete evaluation is the analysis of vulnerabilities of our caching solution. We are aware that caching solutions as presented in the context of this paper have to be secured against attacks. Possible target points for our system may be the cache entries themselves (manipulation, deletion, etc.), the communication channels between the cache, the PEP, the PDP, the context providers as well as the cache manager. Targets may also be the open constraints when evaluated on the cache side as well as potentially manipulated received information from the context provider(s). This will be further analysed in the future and aligned with a proposal of respective measures.

7. Conclusion

In this work we presented a caching solution of access control decisions for business process-driven environments. The proposed caching strategy relies on the fact that the execution of business processes takes place within a small margin of actions which always happen subsequently in predefined order. This makes it possible to pre-compute cache entries in such a way that decision responses are available exactly when needed.

Our approach relies on dependency relations which state exactly at which points during the execution of a business process it is optimal to pre-compute a cache entry as well as for which potentially upcoming event an entry should be created and at which point during execution the entry will no longer be needed and can be removed.

We presented an accompanying architecture. We implemented and benchmarked our proposed solution. The results show that a significant performance gain can be expected if our caching strategy is implemented.

8. References


