A Framework for ProActive Caching in Business Process-driven Environments

Dissertation

submitted to

Technische Universität Darmstadt - Department of Computer Science

for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

presented by

Mathias Kohler, M.Sc.

born in Emmendingen, Germany

Examiner: Prof. Dr. Claudia Eckert, Technische Universität München
Prof. Dr. Max Mühlhäuser, Technische Universität Darmstadt

Submission: 20. May 2011
Oral Examination: 05. July 2011

Published in Darmstadt, 2011 Darmstädter Dissertation D17

Abstract

System response times influence the satisfaction of users interacting with a system. Research shows that increasing response times lead to increasing dissatisfaction or complete refusal of using the system.

System analyses show that enforcing access control requirements significantly influence the system's performance experienced by end users. With increasing regulatory demands such as Basel II, Sarbanes Oxley, or data protection laws, modern complex and multi-layered enterprise systems require fine-grained and context sensitive enforcement of access control policies. Consequently, an efficient policy evaluation is getting more and more important to ensure a satisfactory system performance for interactive tasks.

Research in the area of performance optimizations of access control evaluations is well known, comprising replication of respective system components, structural optimizations of the security policy, as well as different caching strategies. All these approaches have in common that the presented optimization techniques try to optimize access control evaluations independently from the system context.

Modern enterprise systems are inherently based on models for process execution. These models provide a detailed view on the system context and, thus, enable new caching approaches. The dynamic nature of today's process management systems and increasing demand for context sensitive security enforcement, however, challenge caching access control decisions as changing context strongly impacts on the continuous validity of stored access control decisions.

In this thesis, we propose ProActive Caching, a caching strategy specifically tailored to the dynamic properties of business process-driven environments. ProActive Caching aims at providing a significantly low response time for access control decisions, as well as allowing to cache access control decisions which are based on context sensitive security policies.

Moreover, we provide an accompanying caching architecture and a detailed performance analysis of different caching strategies for static and dynamic aspects of access control policies, showing that our strategy significantly improves the performance compared to other approaches for caching access control decisions.

Extended Abstract (German)

Für prozessorientierte Industrielösungen bestehen die gegenläufigen Herausforderungen, die Reaktionszeiten des Systems für Benutzer optimal zu minimieren und gleichzeitig mittels Zugriffskontrollauswertungen, welche die Reaktionszeit signifikant erhöhen, unbefugte Interaktionen zu verhindern. Verzögerte Antwortzeiten führen zu Unzufriedenheit beim Benutzer und können zu vollständiger Ablehnung des Systems führen.

Für die Autorisierung von Interaktionen eines Benutzers mit in Unternehmen häufig eingesetzten prozessorientierten Industrielösungen, wie beispielsweise für Ressourcenplanung (ERP), müssen eine Vielzahl von Zugriffskontrollanfragen ausgewertet und die intendierte Aktion des Benutzers auf ihre Legitimität hinsichtlich der im System festgelegten "Security Policy" überprüft werden. Als Konsequenz werden die Reaktionszeiten auf Benutzereingaben durch die durchgeführten Autorisierungsabfragen signifikant beeinflusst. Verzögerungen im Bereich von 100 ms werden bereits als kleine Störung wahrgenommen, Wartezeiten von mehr als einer Sekunde unterbrechen bereits den Gedankenfluss.

Gerade zusätzliche, rechtliche Anforderungen durch beispielsweise Basel II, Sarbanes Oxley oder Datenschutzgesetze erfordern in modernen, mehrschichtigen Industrielösungen eine feingranulare als auch kontextabhängige Auswertung und Durchsetzung von Security Policies. Konsequenterweise ist es für die Zufriedenheit der Benutzer wichtig, eine auf Geschwindigkeit optimierte Evaluierung von Sicherheitsanfragen einzusetzen.

Zur Optimierung von Zugriffskontrollanfragen gibt es bereits verschiedene Ansätze. Hierzu gehört die Replikation von denjenigen Systemen, welche für die Auswertung von Zugriffskontrollanfragen genutzt werden oder die strukturelle Optimierung von Security Policies. Eine weitere Möglichkeit ist die Nutzung verschiedener Caching Strategien.

Die Literatur bietet eine Fülle von generischen als auch für Zugriffskontrollauswertungen optimierten Caching Verfahren. Die genannten Ansätze haben gemeinsam, dass die jeweilig dargestellten Strategien zwar eine Optimierung der Performance darstellen, diese jedoch den Einbezug des Anwendungskontextes - innerhalb dessen die jeweilige Strategie eingesetzt werden soll - nicht berücksichtigen.

Gerade der Einsatz von dynamischen Kontextinformationen erhöht die Komplexität von Zugriffskontrollauswertungen. Während bei der Auswertung von statischen Security Policies keine weiteren Informationen über das Anwendungssystem benötigt werden, müssen bei dynamischen Zugriffskontrollauswertungen Systeminformationen hinzugezogen werden. Diese werden benötigt, um beispielsweise Entscheidungen zur Einhaltung des Vier-Augen-Prinzips durchzusetzen; dies ist jedoch nur möglich, wenn bei der Auswertung bekannt ist,

ob der jeweilige Benutzer zuvor bereits sich ausschließende Interaktionen am System oder einem Geschäftsobjekt durchgeführt hat.

Diese Einbeziehung von sich dynamisch veränderbaren Kontextinformationen erschwert den Einsatz von Caching Lösungen. Die Veränderung einer zur Zugriffkontrollauswertung genutzten Information führt unweigerlich dazu, dass zuvor ermittelte Zugriffskontrollentscheidungen ungültig werden können. In der Konsequenz können Zugriffskontrollentscheidungen nur in einem Cache gespeichert werden, wenn sichergestellt wird, dass ausschließlich gültige Einträge aus dem Cache abgerufen werden können.

Diese Doktorarbeit stellt eine Caching Strategie vor, welche speziell für den Einsatz in prozessorientierten Industrielösungen entwickelt wurde. Dabei werden zwei wesentliche Ziele verfolgt:

- Das erste Ziel ist eine signifikante Reduktion für die Bereitstellung von Zugriffskontrollentscheidungen, welche durch ein prozessorientiertes System verarbeitet und durchgesetzt werden können.
- Das zweite Ziel ist eine Cache-Management Strategie, welches auch die Speicherung von solchen Zugriffskontrollentscheidungen erlaubt, welche neben der Berücksichtigung der Security Policy mittels dynamisch veränderbaren Kontextinformationen ausgewertet wurden.

Diese Ziele erreichen wir mittels der folgenden fünf technischen Beiträge:

Erstens durch "ProActive Caching" (PAC). PAC ist eine Caching-Strategie, die speziell für den Einsatz in Geschäftsprozess unterstützen Unternehmen entwickelt wurde. Sie ist speziell darauf ausgelegt die Bereitstellungen von Zugriffskontrollentscheidungen signifikant zu beschleunigen. PAC ermöglicht es hierfür, Zugriffskontrollanfragen anhand vorhandener Prozessmodelle vorauszuberechnen und zwischenzuspeichern, sodass Zugriffskontroll-antworten direkt aus einem Cache beantwortet werden können. Dies gilt selbst für "Erstzugriffe", welche bei regulären Caching Strategien üblicherweise nicht aus dem Cache beantwortet werden können. Darüber hinaus ermöglicht PAC die Vorhaltung solcher Zugriffskontrollentscheidungen, welche von bestehenden Caching Strategien nicht gespeichert werden können, da die Entscheidung über die Zugriffserlaubnis auf dynamisch verändernden Kontextinformationen beruht.

Zweitens wird in der vorliegenden Arbeit eine hybride "2-Level Caching Strategie" definiert, welche es ermöglicht, PAC mit weiteren Caching Strategien zusammen zu verwenden. Die Vorausberechnungen von Zugriffskontrollentscheidungen bedeuten für PAC einen größeren Overhead für das Cache Management, wie dies bei alternativen Caching Strategien der Fall ist. Alternative Strategien können jedoch keine auf dynamischen Kontextinformationen beruhenden Zugriffskontrollentscheidungen speichern. Durch die gemeinsame Nutzung von

PAC mit alternativen Caching Methoden können Vor- und Nachteile jeweiliger Strategien kompensiert werden.

Drittens werden Lösungen zur automatischen Generierung einer Caching Heuristic präsentiert, welche eine Voraussetzung sind, um Zugriffskontrollentscheidungen vorauszuberechnen. Mittels der Caching Heuristic wird definiert, zu welchem Zeitpunkt während der Ausführung eines Geschäftsprozesses Zugriffskontrollentscheidungen vorausberechnet werden. Insbesondere auch, auf welcher Basis für Zugriffskontrollanfragen dies geschieht und zu welchem Zeitpunkt bereits gespeicherte Zugriffskontrollentscheidungen nicht mehr benötigt werden und somit aus dem Cache entfernt werden können.

Viertens wird empirisch die Performance von PAC in Geschäftsprozessumgebungen analysiert und mit den Ergebnissen der Analyse alternativer Caching Strategien verglichen. Dies beinhaltet auch die Performanceanalyse des zuvor genannten hybriden Ansatzes.

Fünftens wird eine allgemein einsetzbare Caching Architektur für die Anwendung von PAC in Geschäftsprozess-unterstützten Unternehmen beschrieben. Die Architektur ist jedoch nicht auf PAC beschränkt, vielmehr erlaubt sie allgemein die Integration von alternativen Caching Strategien.

Insgesamt stellt diese Arbeit eine Lösung zur zeitoptimierten Bereitstellung von Zugriffskontrollentscheidungen vor. Die empirischen Ergebnisse zeigen, dass PAC die Antwortzeiten für Zugriffskontrollanfragen gegenüber einer Nicht-Optimierung als auch im Vergleich zu alternativen Caching Strategien signifikant reduziert. Gerade im Umfeld von Geschäftsprozessen mit häufiger Interaktion der Anwender in IT-Systemen sind schnelle Reaktionen auf Benutzereingaben elementar. Mit PAC können Zugriffskontrollanfragen nahezu direkt beantwortet werden.

Acknowledgements

I am particularly thankful to my supervisor, Prof. Dr. Claudia Eckert, for her continuous support throughout this project and for her insightful discussions and valuable advice. My sincere gratitude also goes to Prof. Dr. Max Mühlhäuser for evaluating my thesis as second examiner.

My special thanks go to my supervisors Dr. Andreas Schaad and Dr. Achim Brucker for their constant supervision of this project on part of SAP Research.

My particular appreciation also goes to Prof. Dr. Günter Müller as well as to my colleagues at SAP Research for our active discussions and in this way contributed opinions and suggestions.

Last but not least, I would like to thank my parents for their confidence in my abilities and their constant support throughout this endeavour.

Contents

1 **Introduction** 17
1.1 Motivation 17
1.2 Method of Approach 20
1.3 Goals and Contributions 22
1.4 Thesis Structure 23

2 **Business Process Management** 27
2.1 Business Process Definition 27
2.2 Business Process Execution 34
2.3 Summary 50

3 **Access Control for Business Process-driven Environments** 51
3.1 Security Policy: Permissions & Constraints 51
3.2 Caching Access Control Decisions 58
3.3 Reference Architecture for Access Control in Process-driven Systems 60
3.4 Summary 64

4 **Related Work** 65
4.1 Caching & Prefetching 65
4.2 Caching Access Control Decisions 67
4.3 Performance Improvements within the Area of Access Control 70
4.4 Summary 72

5 **ProActive Caching Strategy** 73
5.1 Overview 73
5.2 Strategy 74
5.3 Foundations 81
5.4 Pre-evaluating, Caching and Updating Dynamic Access Decisions 95
5.5 Summary 104

6 **Caching Heuristic** 107
6.1 Preliminaries 107
6.2 Static Access Control 109
6.3 Generation of General Revocation Triggers 117
6.4 Business Objects 118
6.5 Dynamic Access Control 120
6.6 Summary 124

7 **Aspects for Optimization** 125
7.1 Reusing the Cache for Access Control Pre-Evaluations 125

7.2 Hybrid Caching Architecture (2-Level Cache) 130
7.3 Summary 132

8 Analysis 133
8.1 Functional Classification of Caching Approaches 133
8.2 Environmental Setup 134
8.3 Static Access Control 136
8.4 Dynamic Access Control 144
8.5 Summary 152

9 Conclusion & Future Research 155
9.1 Conclusion 155
9.2 Discussion on Updating Cache Entries 157
9.3 Future Research 158

10 References 161

11 Appendix - EPC to BPMN Transformation 167

List of Abbreviations

ANSI	American National Standard Institute
BO	Business Object
BoD	Binding of Duties
BoD-DR	constraint-specific dependency relation for Binding of Duties
BPM	Business Process Management
BPMN	Business Process Modeling Notation
BPMS	Business Process Management System
CRM	Customer Relationship Management
DCard-DR	constraint-specific dependency relation for Dynamic Cardinality
DR	Dependency Relation
DSoD	Dynamic Separation of Duties
DSoD-DR	constraint-specific dependency relation for Dynamic Separation of Duties
ERP	Enterprise Resource Planning
extDR	extended Dependency Relation
GRT	General Revocation Target
GWL	General Worklist
jBPM	name of the workflow engine of JBoss
LC	Life cycle
OC	Open Constraint
ORKA	Organisational Control Architecture
PAC	ProActive Caching
PD	Process Definition

PDP	**Policy Decision Point**
PEP	**Policy Enforcement Point**
PIID	**Process Instance Identifier**
PLC	**Process Life Cycle**
RBAC	**Role-based Access Control**
RT	**Revocation Target**
SAAM	**The Secondary Approximate Authorization Model**
SC	**Standard Caching**
SCM	**Supply Chain Management**
SoA	**Service-oriented Architecture**
SP	**Security Policy**
SR	**Successor Request**
SSoD	**Static Separation of Duties**
TIID	**Task Instance Identifier**
TLC	**Task Life Cycle**
TT	**Trigger Target**
UWL	**Universal Worklist**
WfMC	**Workflow Management Coalition**
WfMS	**Workflow Management System**
XACML	**eXtensible Access Control Markup Language**

List of Definitions

Definition 1 Event 44
Definition 2 Formal Life Cycle Model 45
Definition 3 Dynamic Context Constraints 59
Definition 4 Change-detectable Context Constraints 60
Definition 5 Open Constraints 60
Definition 6 Access Control relevant Event 61
Definition 7 Cache Entry 80
Definition 8 Trigger Target 86
Definition 9 Successor Request 87
Definition 10 Dependency Relation 88
Definition 11 General Revocation Target 91
Definition 12 Revocation Target 93
Definition 13 extended Dependency Relation 93
Definition 14 Dynamic SoD Dependency Relation 98
Definition 15 Binding of Duties Dependency Relation 100
Definition 16 Dynamic Cardinality Dependency Relation 101
Definition 17 Open Constraint-aware Cache Entry 103
Definition 18 Access Control Aware Formal Life Cycle Model 108
Definition 19 Cross-instance Successor Request 127

1 Introduction

In this chapter, we motivate the topics of this doctoral thesis. In Section 1.1 we first describe the motivation of our work and discuss the problems addressed. Section 1.2 gives an overview about our method of approach. In Section 0 we present our main contributions. Finally, Section 1.4 closes this chapter with a structure of this thesis.

1.1 Motivation

The influence of system response times on the satisfaction of users interacting with the system is well known since more than 30 years [38]. Practical experience shows that already delays of 100 ms are perceived as an interruption, while a delay of more than 1 second impacts on the flow of thoughts [42]. This is further supported by experimental research showing that increasing response times and slow system reactions lead to increasing dissatisfaction of users or even complete refusal of using the system [24]. Hence, it is very important that systems, including enterprise systems for business process management, provide a satisfying performance.

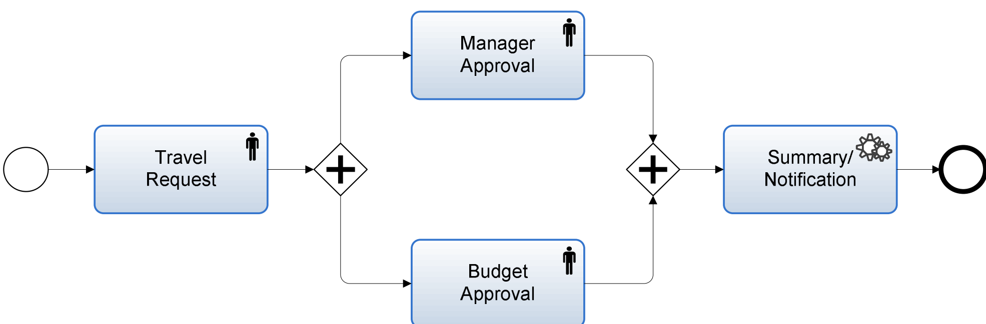


Figure 1-1: Travel request process with three single tasks

Enterprises more and more rely on business process management systems (BPMS) which provide the execution of complex business processes. Business processes describe a flow of tasks which have to be executed in a pre-defined order such that a related business objective is achieved. Business processes, e. g., for enterprise resource planning (ERP), supply chain management (SCM), or customer relationship management (CRM) are important for today's enterprises to improve operational efficiency, consistency, and quality [12, 20]. The majority of enterprises already implement business processes [19]; more than 70% of all economic processes are already executed with IT support [48].

Figure 1-1 illustrates a travel approval process of a fictive enterprise. This process comprises four tasks, namely "Travel Request", "Manager Approval", "Budget Approval", and

"Summary/Notification". In the first step, the employee enters the dates, estimated costs, and an additional description for the travel which he wants to request into a process management system. Afterwards, this request is transferred to two managers which both have to approve the request. One of them is the employee's manager, the other one is a manager responsible for travel budgets. Each of the managers marks the request as approved or not approved. In the fourth step, the system automatically evaluates the managers' approvals and the employee receives a final notification with the outcome of his request. With the last step, the process ends.

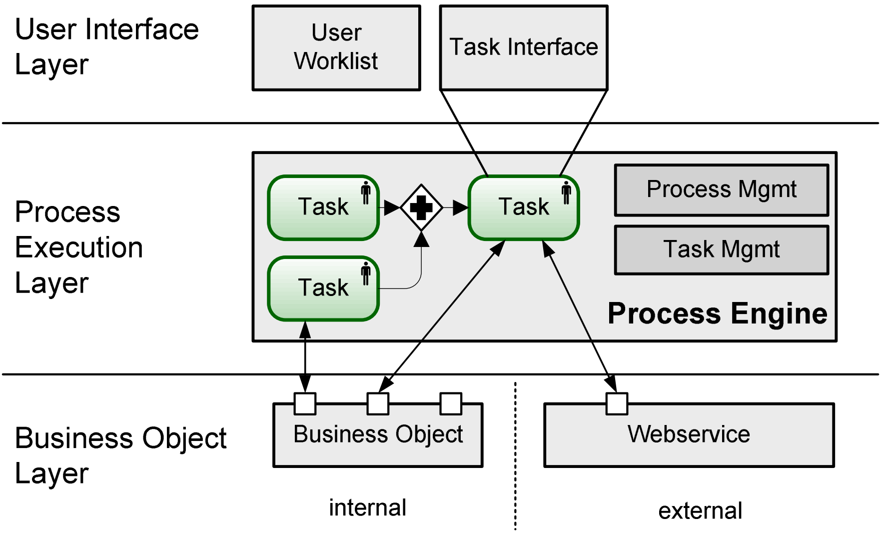


Figure 1-2: Generic architecture illustrating the three layers for business process execution

BPMSs allow complex interactions between a user and internal processes and applications required for the execution of such processes, while their architectures become increasingly extended with additional capabilities for B2B (business-to-business) interactions between different companies, as well as the support for service oriented architectures (SOA) [20]. This leads to an execution of business processes over multiple layers. The different layers can be considered as depicted in Figure 1-2, comprising a *user interface layer* providing a user an interface for process executions, a *process execution layer* for business process management, as well as a *business object layer* for accessing internal back-end applications or external web service providers.

Large enterprise systems increasingly contain a large amount of confidential data (e.g., financial data, or customer data) which are administrated and processed by BPMSs. Regulatory requirements such as Basel II [2] or Sarbanes Oxley [21, 53], as well as data protection laws require complex access control restrictions on such systems. This leads to the fact, that enterprise systems implement mechanisms for the enforcement of fine-grained, complex and dynamic access control policies.

Due to the open architecture of service oriented architectures and the possibility to access each layer individually, access control enforcement needs to be made at all different layers on which business processes are executed. Moreover, on a vertical level, business requirements driven by regulatory demands, result in an increase in the amount and complexity of access control policies for business process executions. This includes the need for checking the current system state, as well as checking an increasing number of user interactions against the system's security policy. Checking the security policy allows to evaluate an access control decision, stating whether the current user request for interaction may be granted or must be denied.

Due to the dynamic nature of business process executions and the requirement that security policy evaluations must take the current system status, as well as external factors into account, the overall system performance, and the performance eventually experienced by the user is significantly affected.

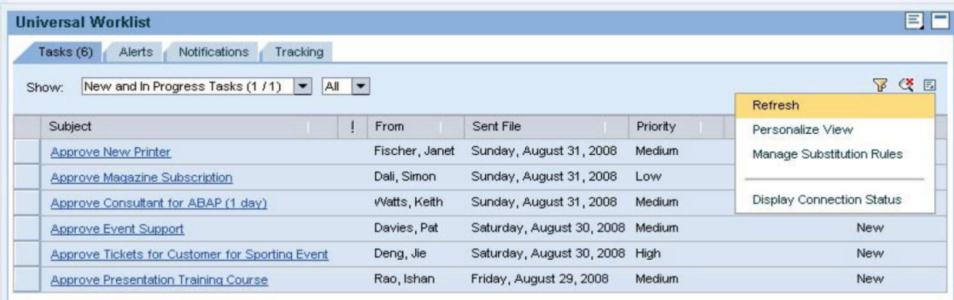


Figure 1-3: Example of a Worklist of an SAP NetWeaver BPM where a user has access to all tasks available to be executed [51]

The *General Worklist* (GWL), for instance, is the main interaction point for user-centric execution of business processes. Figure 1-3 shows a GWL[1], presenting a list of tasks which are currently waiting to be executed by a user. The user selects and claims the tasks he feels responsible for and subsequently executes the tasks.

Usually, a user may only claim a small fraction of all tasks available in the system. Tasks may not be claimable based on a lack of required access rights. These access rights are based on the system's security policy, but may be further influenced by dynamic aspects like the user's previously executed tasks or the current system status.

These dynamic aspects (usually called dynamic context information) may change over time. Different, alternating context information may, however, result in different, alternating access control decisions. Obviously, the GWL should only display those tasks to the user,

[1] also called *Universal Worklist* (UWL) [51]

which he is actually allowed to claim and execute. This requires that the access rights of a user need to be checked each time the GWL is displayed, because the context information allowing a user claiming a task might have changed since he entered the GWL the last time.

The number of tasks in the GWL depends on the number of process instances running in the system. If, for instance, 300 process instances are active, each of them having at least one but possibly multiple active tasks would result in 300+ active instances of tasks. If only 30% of them are in a status ready to be claimed, meaning they would show up in the GWL, access control checks for about 100 tasks must be performed every time a worklist is displayed to a user. Our experiments show that a typical access control check in large enterprise systems requires about 25 ms to 35 ms which results in a delay of at least two to three seconds every time the GWL is displayed.

State of the art industrial BPMSs execute hundreds of thousands of business process tasks in parallel, making the efficient implementation of the enforcement and evaluation of security policies increasingly important.

1.2 Method of Approach

State of the art security infrastructures are already implemented with efficiency in mind [34, 57]. Moreover, there is a wealth of literature [6, 8, 15, 27, 34, 35, 39, 68, 69] which deals with performance optimization of access control evaluations, introducing several approaches to improve the response time for requests on access control decisions.

Access control enforcement requires a security policy based on which access control requests are evaluated. IBM's Tivoli Access Manager [27], for instance, uses replication of such a security policy to optimize the access control request evaluation in distributed systems. Another approach addressed for performance improvements is to transform the structure of the security policy. Goal is to transform the policy into a form which reduces the required processing time to compute an access control decision [34, 35, 39]. Other approaches discuss both, generic caching strategies [11, 26, 36], i.e., strategies which are independent of the application area, and access control specific caching strategies [8, 15, 68] for improving the response times for access control requests.

All these approaches have in common that the presented optimization techniques try to optimize access control evaluations independently from the system context. Especially in dynamic environments as found for business process executions, access control evaluations inherently depend on dynamic context information which may constantly change its current values. Currently, each access control evaluation requires fetching the current system context such that access control decisions take the lasted context changes into account. Optimization techniques are required to adapt to such new challenges.

In consequence, optimizations on policy transformations may only bring partial improvements, as the data for the current system context must still be fetched with each evaluation of an access control request. Standard caching strategies, as well as more sophisticated approaches, either completely lack the possibility to cache access control decisions based on the evaluation of dynamic context information (e.g., SAAM [15, 68]), or try to model invalidation techniques into cached decisions objects which may render cache entries invalid, given an exceeding timeout or threshold value (e.g., CPOL [8]). Moreover, none of them makes use of the underlying process and workflow models.

Modern enterprise systems, however, are inherently based on such process and workflow models. These model-centric systems enable new approaches for caching strategies improving the performance of security evaluations.

Our strategy exploits the fact that executions of business processes are based on the execution of tasks in a partially pre-defined order. We take advantage of this fact and evaluate relevant access control decisions upfront for subsequent tasks of a currently executed process.

The integration of business process information about the possible system behaviour allows anticipating future access control requests 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 caching strategy is to our knowledge the first workflow specific caching strategy which exploits the business process and workflow models for providing instant access control decisions and avoiding cache misses by systematically calculating these decisions upfront.

Moreover, our strategy provides solutions to handle access control decisions which are based on dynamic context information. Such access control decision may become invalid over time if the respective context information changes. Our caching heuristic explicitly allows defining rules for updating such cached decisions, enabling caching even for dynamic and complex environments such as BPMSs.

1.3 Goals and Contributions

The research objective for our work presented in this thesis is the development of a framework for caching access control decisions within business process-driven environments. We specifically aim at the following two goals:

- Our first goal is significantly reducing the time currently required until access control decisions can be consumed and enforced by the respective BPMS.
- Our second goal is a cache management strategy that allows to cache access control decisions which are based on security constraints relying on dynamic context information.

For achieving these goals, we provide the following five technical contributions.

ProActive Caching

Firstly, we present *ProActive Caching* (PAC) [29-31], a caching strategy which is specifically tailored to the dynamic properties of business process-driven environments. The strategy is to improve the system's performance, especially for user-centric tasks.

PAC allows answering all access control requests directly from the cache, rather than regularly evaluated by an access control decision component. Hence, PAC optimizes the availability of cache entries and enables providing answers even for first-time queries. The idea is that getting an access control decision from the cache is by magnitudes faster than waiting for a standard, regular request evaluation.

Furthermore, PAC enables to cache access control decisions based on security constraints which rely on dynamic context information.

Hybrid caching strategy

Secondly, we present a novel, two-levelled caching strategy allowing the combination of our PAC strategy for dynamic access control policies with strategies for static access control policies.

PAC is capable of dealing and caching access control decisions which are based on the evaluation of dynamic context information. Its pre-evaluation allows for answering all access control requests directly from the cache - even on their first occurrence. These pre-evaluations, however, put a higher overhead onto the system than standard caching strategies which only rely on regular access control evaluations, storing the resulting decisions in case they are required a second time. The drawback of standard caching approaches is their lack of support for caching access control decisions based on dynamic context.

Our two-levelled cache approach allows the combination of strategies such that drawbacks from one strategy can be compensated by the strengths of another one, and vice versa.

Automatic Generation of Caching Heuristic

Thirdly, we present means to automatically generate a caching heuristic required to provide, in a generic way, instructions for the management of our cache.

The caching heuristic defines which access control requests should be pre-evaluated, the point during a process execution the pre-evaluation should be triggered, as well as the point during the process execution where cached access control decisions are not needed any longer and may be revoked from the cache.

Analysis and Comparison

Fourthly, we analyze and present performance tests for PAC and compare them to other caching strategies. Especially for system designers, metrics for comparing different caching strategies are a prerequisite for allowing to choose the optimal solution for a specific system. Both, the performance and the cache size, in relation to different process types and security policy types, provide metrics for comparing access control caching strategies.

Moreover, we present performance results on the combination of PAC with the other caching strategies using our hybrid caching approach.

Generic Caching Architecture

Finally, we present a generic caching architecture for business process-driven environments. In large enterprise systems access control enforcement usually follows the request-response paradigm [27, 44, 46]: an access control evaluation component answers access control requests from one or several application-specific access control enforcement components. Given the three layers we presented above (cf. Figure 1-2), we present an extended standard architecture for enterprise systems with a caching component, enabling caching of access control decisions. This architecture does not depend on the implemented caching strategy: generic caching strategies can be as easily integrated as access control specific ones (e.g., [30, 68]).

1.4 Thesis Structure

This thesis is structured in nine chapters. It starts with an overview about business process management, access control for business process-driven environments, and related work. This is followed by the introduction of our caching strategy and accompanying caching heuristic, as well as dealing with aspects of optimizing the handling of cached decisions. Finally we provide a performance analysis and conclude with a summary, discussion and outlook for future work.

Chapter 1, *Introduction*, gives the motivation of this thesis, introduces our goals, method of approach, and describes our main contributions.

Chapter 2, *Business Process Management*, introduces the context of business process execution. This chapter describes background information about business process definitions, business process execution, and the respective business process management systems. It builds the foundation to support the understanding of the upcoming chapters and introduces common, important terms used within the context of business process-driven environments.

Chapter 3, *Access Control for Business Process-driven Environments*, provides a brief introduction into access control enforcement for business process-driven systems. This chapter describes the requirements such dynamic process-driven systems impose on enforcing access control evaluations and which challenges this imposes on caching strategies for such systems. Furthermore, the enforcement of access control for business process executions requires an architecture supporting the evaluation and application of access control policies. Within this chapter, an access control reference architecture specifically developed for business process-driven systems is introduced, functioning as basis for our work. This reference architecture will later be extended for enabling our caching strategy.

Chapter 4, *Related Work*, discusses related work. We look into work dealing with pre-fetching required information, work dealing with caching of access control decisions, as well as work which uses other approaches to improve the performance within the area of access control.

Chapter 5, *ProActive Caching Strategy*, introduces our caching strategy by giving a complete description about the idea of ProActive Caching, the underlying caching heuristic, the information sources required for creating the caching heuristic, as well as an introduction of the additional components required for our reference architecture. Moreover, two different approaches how to deal with access control decisions relying on dynamic context information are introduced in detail.

Chapter 6, *Caching Heuristic*, illustrates in detail how the caching heuristic for our caching strategy can be generated automatically based on the information sources presented in the last chapter. All algorithms required are respectively introduced.

Chapter 7, *Aspects for Optimization*, describes two modes of operation such that ProActive Caching may be applied in an optimized way. The first aspect introduces means to re-use already pre-evaluated access control decisions across multiple process executions. The chapter discusses to what extend such a re-use is possible and which aspects for storing access control decisions across instances of process executions must be considered.

The second aspect describes a hybrid caching approach which is based on a two-levelled caching architecture, where each level contains a different caching strategy, enabling a combination of ProActive Caching with other caching strategies. This specifically allows choosing strategies in such a way that if combined, drawbacks from each single strategy can be compensated.

Chapter 8, *Analysis*, presents and analyzes ProActive Caching according to performance improvements and provides a detailed comparison with other approaches for caching role-based and dynamic access control decisions in business process-driven enterprise systems. In particular, we compare generic caching strategies against caching strategies developed to specifically cache access control decisions. Furthermore, we report on the performance of our hybrid caching approach.

Chapter 9, *Conclusion & Future Research*, summarizes and discusses the results of our work and identifies lines of future research.

Associated Publications

Parts of our work, as well as underlying ideas have been previously published in the following forms:

- Mathias Kohler, Christian Liesegang, Andreas Schaad: *Classification Model for Access Control Constraints.* In proceedings of the 26th International Performance Computing and Communications Conference (IPCCC), 2007.
- Mathias Kohler, Andreas Schaad: *ProActive Access Control for Business Process-driven Environments.* In proceedings of the 24th Annual Computer Security Applications Conference (ACSAC), 2008.
- Mathias Kohler, Robert Fies: *ProActive Caching - A Framework for performance optimized Access Control Evaluations.* In proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY), 2009.
- Mathias Kohler, Achim D. Brucker, Andreas Schaad: *ProActive Caching: Generating Caching Heuristics for Business Process Environments.* In International Conference on Computational Science and Engineering (CSE), IEEE Computer Society, 2009.
- Mathias Kohler, Achim D. Brucker: *Access Control Caching Strategies - An Empirical Evaluation*. In the 6th International Workshop on Security Measurements and Metrics (MetriSec), 2010.

2 Business Process Management

In this chapter we introduce the reader into the context of business process management (BPM). We will give background information about business process definitions, business process execution, and respective business process management system (BPMS). This information is to support the understanding of the upcoming chapters and introduces common, important terms used within the context of BPM.

The content of this chapter is structured as follows. Firstly, we concentrate on business processes, how they are defined and what they are used for. Secondly, we concentrate on business process execution. We will describe a generic system architecture and illustrate how process execution happens on three different layers. Thirdly, we go into the details of a specific part of business process execution, namely the introduction of process and task life cycles, as well as system events.

2.1 Business Process Definition

Enterprises often have the business objective to either produce goods or to provide services to their customers. Means and execution instructions to achieve these objectives are often described as *Business Processes*. Business processes serve a business objective by defining the processes and tasks necessary for its achievement, as well as the interaction with systems and (human) resources. Commonly known business objectives are, for instance, "purchase order" or "invoice processing" [16]; often there are additional area specific objectives such as "granting a loan to a customer" in the financial area [54], or "changing a federal law" in the area of government agencies [55].

The Workflow Management Coalition (WfMC), a standardizing body with the goal to establish workflow technologies and standardized interfaces, puts the definition of a business process in similar words, adding that the execution of business processes might be put in context of organizational structures of an enterprise: "A set of one or more linked procedures or activities which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles and relationships." [72]

Business processes (or simple processes) can be implemented and technically supported by one or multiple connected *Workflows*. Workflows describe a flow of tasks which have to be executed in a pre-defined order such that the related business objective is achieved. The goal of implementing such workflows is to increase efficiency, consistency, and quality [12].

Besides the already stated business objectives above, business process implementations are typically used for approval processes (e.g., purchase requisition, purchase order, or invoice approval), administrative processes (e.g., time booking, process to hire new employee, or travel reimbursement), processes which must strictly be executed for auditing purposes (e.g., for areas where a proof for Sarbanes Oxley compliance [53] is required), or production processes (e.g., processing customer orders) [41].

In our work we mainly concentrate on the *execution* of business processes and will therefore mostly refer to their structural aspects, i.e., their workflow representations. Hence, if not stated otherwise, the words 'process' and 'workflow' are used interchangeably.

In general, there are well known collections of proposed standard processes. SAP [52], for instance, published several standard processes with its SAP R/3 system; some of them can be found in standard textbooks (e.g., [16, 56]).

2.1.1 Process Definition Languages

Processes have different sizes, depending on the number of tasks. In principle, there is no limit of how many tasks a process may contain; in our experience the number, however, ranges from a few up to a couple of dozen tasks. The modelling of processes is usually done by using one of the business process definition languages which have been established over recent years.

Business Process Modeling Notation (BPMN) [45] is a quite powerful and up-to-date language to define business processes. We will use BPMN to illustrate examples and descriptions given in our work. This requires a short introduction into the modelling language concepts of BPMN which we will give next. We will keep this on a level, such that it is sufficient to understand the examples and illustrations given in this work. For more in-depth details about the language and its structural elements, the interested reader is referred to the BPMN specification document [45].

Processes have start events and end events. They clearly define at which point(s) the process execution starts and at which point(s) it ends. One possible start event is, for instance, that a customer wants to buy a product and initiates an internal process by submitting a contact form, leading to an internal message which triggers the process execution. There are other start events, like timer events, or conditional events. In BPMN, a start event is denoted as an empty circle if no trigger is defined, and as a circle including a symbol for start events with triggers. The few named examples are illustrated in Table 1.

Trigger	Description	Marker
None	Marker is used if no specific trigger is defined.	
Message	A received message triggers the execution of a process.	
Timer	A specific time or date is set to trigger the execution of a process.	
Conditional	A condition is set to specify that, if a certain threshold is reached, the execution of a process is triggered.	

Table 1: Selected BPMN Start Events [45]

Similar to start events, BPMN also defines end events which finish a process execution. A process might just terminate if all tasks have been finished, it can "throw" a message notification, or uses error signalling. Table 2 shows selected end events as examples.

Trigger	Description	Marker
None	This marker is used if there is no specific result.	
Message	This end marker indicates that with the end of a process execution a message is sent to a receiver defined within a process.	
Error	This end marker indicates that an Error handling should be triggered.	

Table 2: Selected BPMN End Events [45]

Processes define a set of tasks which have to be performed in a certain pre-defined order. Tasks can generally be divided into two types: automated tasks and human tasks. The first type of tasks is automatically executed by a BPMS. A *Service Task* is an example for an automated task. Service tasks automatically call web services and process the web service's answer to complete the task. The second type of tasks, human tasks, involves users which have to interact with the system to complete the task.

Recall the travel request process mentioned in Chapter 1: the first three steps are human tasks, where the employee and his managers have to give their inputs such that the process can proceed. The last task is an automated task; it sends an automated notification, which does not require any further interaction with a person.

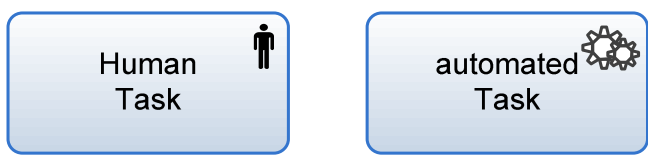
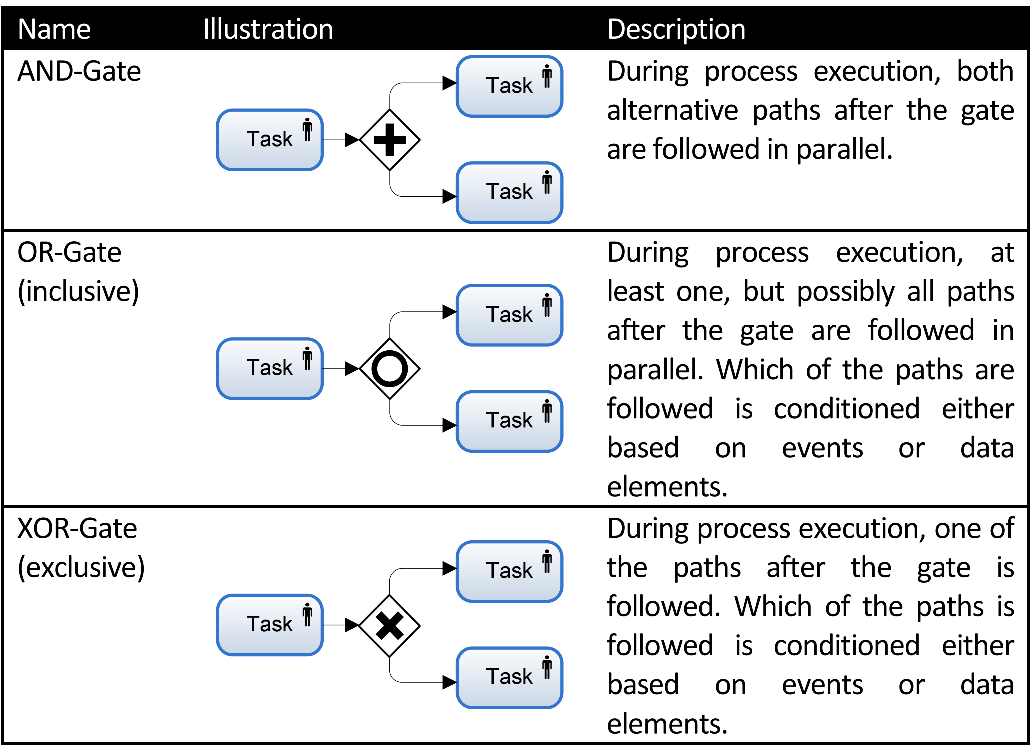


Figure 2-1: Graphics for Human Tasks and automated Tasks

Graphically we distinguish between automated tasks and human tasks by small symbols on the right top corner of a task. Figure 2-1 illustrates this. In this work we concentrate on processes which have a majority of human-interactions involved.

The order the tasks of a process are executed is defined by the order in which the tasks of a process are connected. Control flow elements such as the arrows used to connect the tasks of the travel request process given with Figure 1-1, mean intuitively that after the execution of one task, the execution of the connected tasks can start; hence, after the "Travel Request" has been submitted, the two approval-tasks follow.



Name	Illustration	Description
AND-Gate	Task → AND gate → Task, Task	During process execution, both alternative paths after the gate are followed in parallel.
OR-Gate (inclusive)	Task → OR gate → Task, Task	During process execution, at least one, but possibly all paths after the gate are followed in parallel. Which of the paths are followed is conditioned either based on events or data elements.
XOR-Gate (exclusive)	Task → XOR gate → Task, Task	During process execution, one of the paths after the gate is followed. Which of the paths is followed is conditioned either based on events or data elements.

Table 3: Selected structural control elements in BPMN: AND, OR, and XOR gates [45]

Processes can be of linear and non-linear type. Linear processes have multiple tasks, which are executed in a sequential order. Non-linear processes have more complex structures, including branches and loops such that multiple tasks may be executed in parallel. The travel request process mentioned above is an example for a non-linear process with tasks executed in parallel. To realize non-linear process structures, different kinds of control elements such

as AND-gates, OR-gates, and XOR-gates (see Table 3 for descriptions based on the process language BPMN) can be used. These elements allow that a process execution branches at one point during its execution and merges again at a different point.

For OR-gates and XOR-gates additional conditions are defined with the process, such that during a process execution the *process engine* selects the appropriate path. A very simple condition may state that the path to be followed depends on the input the user made when performing the task right before the branch. A manager, for example, has to approve or decline a purchase request of one of his employees. Depending on the manager's decision, the process either follows the path which cancels the request and sending a notification to the employee that the manager declined the request, or the process follows the path to trigger a new purchase order.

Control elements allow that the actual path during the execution from start events to end events may be different between several executions of the same process.

2.1.2 Business Process Example - Project Issue Management Process

In this section we present an exemplary business process called "Project Issue Management" which is taken from [59]. Figure 2-2 shows the modelled process. The process will be used for a more detailed discussion of the architecture of a BPMS including user interactions and back-end functionality (cf. Section 2.2) as well as how security measures for access control are integrated (cf. Chapter 3).

A full description of the process can be found in [59]; we summarize the purpose of the process and its tasks in the following.

The process gives an example how to deal with issues occurring during the execution of a project (e.g., a large manufacturing project). The motivation of the process is that during the execution of a project conditions, design, scope, and other features are subject to change. This might result in an adapted need of material which has to be ordered and paid, impacting the costs and schedule of the project.

The goal of the process is to provide means to organize the handling of occurring issues during the execution of a project by establishing a change management. The idea is that an organized issue handling provides means to monitor and document such issues and problems, but also manages the updates of budget and schedule which might become necessary. The process execution provides the capability to directly inform and address the person in charge and to handle the specific details until the project issue is resolved.

A process usually starts with a start event. In this case the start event is that a project member recognizes an issue during a project execution. The project member enters a description of this issue into an electronic form and submits it. By entering and submitting

this data a new process instance of the "Project Issue Management" process is created and kicked off.

In the first task "Create Change Request" the project manager has to review the described issue and formulates a change request to resolve it. The project manager finishes her task by submitting the request, including her proposed changes. In the following, this change request is routed to three specialists (purchaser, controller, and scheduler) who now have the tasks to review the proposed changes and analyse the impact the change would cause on the overall project. The process defines three different tasks for this analysis which can be executed in parallel.

Firstly, the process defines the task "Enter Purchasing Data". In this task the purchaser receives the change request the project manager created. His function is to add the data for additional material required to realize the proposed change.

Secondly, the process defines the task "Enter Budget Data". Here the controller enters the remaining project budget into the change request.

Thirdly, the process defines the task "Enter Scheduling Data". In this task the scheduler enters the deadlines at which the materials have to arrive at the latest.

As soon as all of them have been finished the change request is updated accordingly and the project manager must review the updates. The manager also has to decide now whether the current solution should be further followed or not. Hence, in the task "Project Manager Decision" the manager has three options: to submit the request and proceed with the proposed changes, to send it for rework to the three specialists, or to decide to completely cancel the change request.

If the project manager decides to submit it, the business process management system checks two conditions defined with the process. Depending on whether the checks on these conditions evaluate to true or false, further tasks are activated and must be executed. Given the process definition language BPMN, conditions are joined with the structural elements such as the OR-gates following the task "Project Manager Decision" in Figure 2-2.

The first condition is a rule to decide whether a budget manager must approve the proposed changes, given the price the additional material costs (which have been entered by the purchaser) and the information about the remaining budget (entered previously by the controller). If one or more pre-defined thresholds are reached, the budget manager must decide whether the additional required budget is approved or denied. If the budget is denied, the process execution goes into a loop and the project manager is involved again. She has to decide whether the earlier proposed changes have to be reworked or the change request should be cancelled totally.

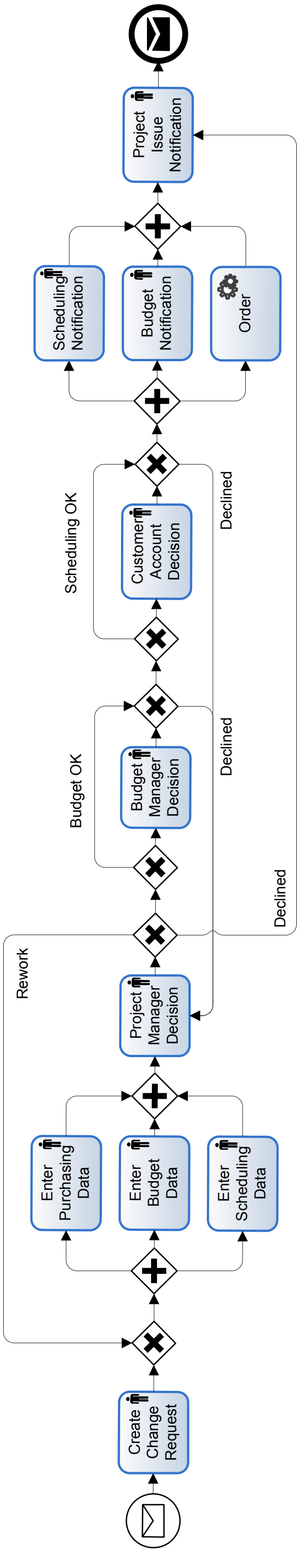


Figure 2-2: Process Model for Project Issue Management [59]

If the first condition results in the case that the budget manager has not to be involved or the budget manager accepts the changes, condition two is checked. In this condition the "material delivery date" and the "project deadline" are compared. If a delay can be expected, the customer must be involved to get its approval on the possible delay. Again, if the customer does not accept the expected delay, the process execution runs into a loop and the project manager gets involved.

Finally, if all possible actions with the budget manager and the customer are resolved and the project manager did not cancel the requested change, three tasks are activated which settle the change request into action. An automated task "Order" creates a purchase order for the additionally required material. Furthermore, the purchaser is informed of the material purchase and the scheduler is informed about the adapted deadlines such that every specialist can update their project planning.

In a last and final step "Project Issue Notification", the project manager receives a summary of the executed process. This task is also executed if the project manager would have decided to cancel the process at an earlier stage. After reviewing the notification, the process terminates.

2.2 Business Process Execution

In most cases business processes are executed tool-supported by a *Business Process Management System*[2] (BPMS). A BPMS allows the creation, deployment and execution of business processes according to their process definitions. The Workflow Management Coalition gives the following definition for a BPMS. "A system that defines, creates and manages the execution of workflows through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with workflow participants and, where required, invoke the use of IT tools and applications." [72]

Examples for BPMSs are JBoss jBPM [62], Bonita Open Solution [7], or SAP's NetWeaver BPM Process Engine [51].

The following subsections describe business process executions in detail. We first introduce an abstract system architecture for a BPMS which will later be used for illustrating our caching strategy presented in this thesis. Furthermore, the system life cycles and system events for process execution will be introduced and defined.

[2] An often used synonym is *Workflow Management System* (WfMS).

2.2.1 Generic System Architecture

Business process execution happens in three execution layers: the *user interface layer*, the *process execution layer*, and the *business object layer*. The three layers build an abstract model for realizing and understanding the execution of business processes in enterprise systems. We briefly introduced them in Section 1.1 and will go into details of all three layers below, starting with the user interface layer. The illustration given with Figure 1-2 depicts the three layers. (Figure 2-3 unterhalb copies the illustration for better readability).

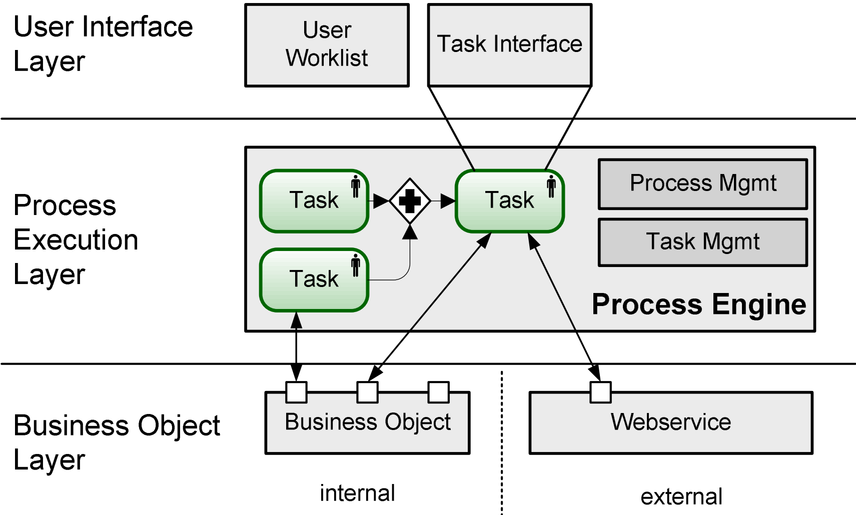


Figure 2-3: Generic architecture illustrating the three layers for business process execution

User Interface Layer

Business process executions may be fully automated, partially automated, or completely user centric. Human based executions comprise the process is user centric and, hence, rely on user interactions to perform single tasks. The responsibility of a BPMS is the process and task management, but a human person is the one eventually claiming and executing the tasks for which he is responsible. In our work, we concentrate on user centric process executions.

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

The first one is the main interaction point of a user with a process management system. It is the *General Worklist* (GWL), also called *Universal Worklist* (UWL) [51]. Before a task instance can be executed, it has to be assigned to a user, called *Resource Allocation*. In today's large enterprises usually more than one user might be applicable to execute certain upcoming tasks. In the "Project Issue Management" process, for instance, multiple project members

might be entitled to calculate a revised project schedule by executing the task "Enter Scheduling Data".

The GWL allows users to specifically claim those tasks currently active in the system for which they feel responsible. The GWL supports this by collecting all active tasks (i.e., task instances) available in the system and rendering them appropriately for the user's selection. The user makes his choice and gets assigned to the selected task. An active task which is already assigned to a user cannot be claimed by any further users. Some process engines (e.g., JBoss jBPM [62]), however, allow a re-assignment of another user to the same task instance such that the first user gets replaced.

Generally, a user may only claim a small fraction of the tasks available in the system. The reason is that for the execution of tasks access rights are necessary. Given a large enterprise, users usually are only responsible for a limited set of the tasks defined in a system. Figure 2-2, depicting the process example for the "Project Issue Management" process, illustrates the relation between roles and tasks by small informal annotations, where each of them states the specific role (such as "Project Manager", "Purchaser", "Scheduler") required for executing the task.

For all these cases the GWL has to perform filtering procedures such that only those tasks are selectable by the user he is actually responsible for and, hence, allowed and able to claim. Whether a user is allowed to claim a task is not only determined according to his roles, but may also be restricted based on previous tasks he already performed. A simple example for the travel request given with Figure 1-1 is that no user should be able to approve his own request. Hence, a user who already performed "Travel Request" should not be allowed to perform the task "Manager Approval" as well. Whether a user gets access to the task "Manager Approval", hence, does not only depend on whether he is assigned to the correct role (Manager in this case), but also whether he already performed the task "Travel Request". This is a dynamic decision which has to be made during runtime. It cannot be decided upfront.

There are other, complex examples on which we will elaborate in Chapter 3. In essence, however, this small example already shows that every time the user accesses the GWL, it has to be checked which of the tasks the user is actually allowed to perform. Especially these checks - whether the user has the correct access rights - cause significant delays in displaying the worklist (as we will show in our performance analysis; cf. Chapter 8); it interrupts the user's flow of work in calling the GWL and waiting for its response.

The number of tasks in the GWL mainly depends on the number of process instances running in the system. Assume 300 process instances are currently active, each of them having at least one but possibly multiple active tasks. This results in 300+ active task instances. If only

30% of them are currently in a status *ready to be assigned*, meaning they would show up in the GWL, access control checks for about 100 tasks must be performed every time a worklist is displayed for a user. In our experiments, an access control check requires about 25 ms to 35 ms, which results in a delay of at least two to three seconds every time the GWL should be called. This does not yet include the time required for the BPMS to select all tasks and render the worklist for displaying it; on larger systems this may take one to three seconds on its own.

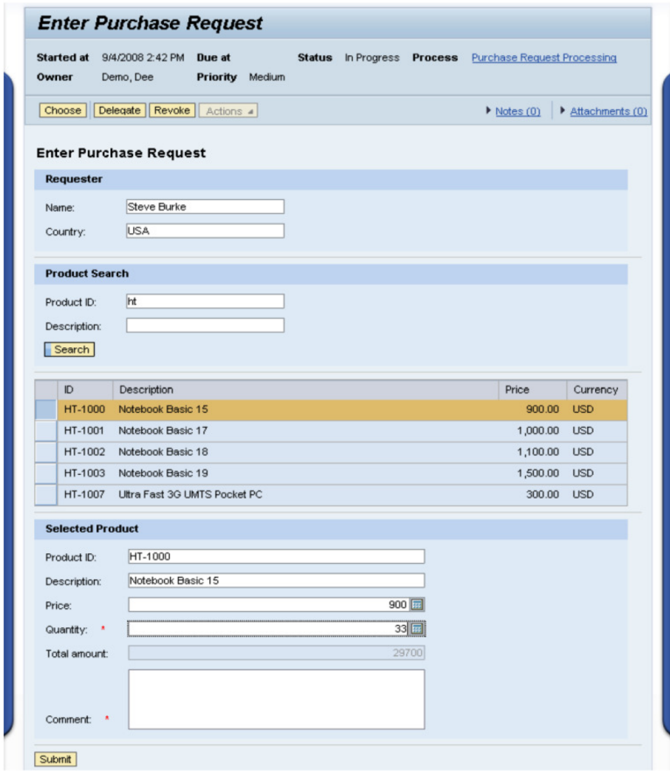


Figure 2-4: Example for a task from in SAP NetWeaver BPM [51]

The work presented in this thesis especially aims at the improvement of the response time for access control checks such that, in case of accessing the GWL, the user gets to see the available tasks instantly. In Chapter 1, we already depicted an example of a GWL with Figure 1-3. It shows the presentation of a list of tasks the user is able to select, claim and execute based on an SAP NetWeaver BPM system [51].

The second type of user interface is the input form for an actual execution of a task. Most human-driven tasks require that a user enters data to complete a task. For these tasks a user interface must be provided with which the user can enter the data the task completion requires. For the "Project Issue Management" process (cf. Figure 2-2) the interface for the first task "Create Change Request" would offer the user the possibility to enter the

description of the requested change as well as possible data for a proposed solution to the issue.

In general, task interfaces may also provide additional information which might help the user to perform the task. In the issue management example, a purchaser performing the task "Enter Purchase Data" must know, for instance, which items are available to purchase. The additional information displayed to a user might be a searchable product catalogue including descriptions and prices for each item available to order. The purchaser can select the items required for the proposed change, add additional comments for later reference, and submit the entered data (see Figure 2-4 oben for illustration).

Process Execution Layer

The definition of processes usually happens in a process designer or some specifically developed modelling tool. After its design phase, the process definition is deployed with a process engine such that the engine can execute it. During runtime, processes are either manually initiated by a user interacting with the system and starting the process, or the engine itself initiates the start of a process up on an event that occurred. Whether a process is initiated manually or automatically is defined within the process definition itself by their respective start events (cf. Section 2.1.1).

For the execution of a process, a new instance of the process is created. In analogy to object oriented programming, this corresponds with the creation of an instance given an object's class definition. A system can create multiple instances of the same process definition such that possibly thousands of process instances may be active in a process engine, running and executed in parallel. The identification of each instance happens by a *Process Instance ID* (PIID) which is usually a system-wide unique number.

During the execution of a process, tasks of the process have to be performed. For each task to be executed, the process engine creates a task instance. Depending on the execution path, the process engine only creates instances of directly subsequent tasks, following the ones currently executed. In fact, the creation of an instance of a task happens if its preceding task has been finished. Furthermore, the process engine considers conditional branching elements (e.g., OR-gates, XOR-gates) such that only task instances are created actually necessary to be executed. For example, the task "Budget Manager Decision" in the previously given process "Project Issue Management" (cf. Section 2.1.2) is only instantiated if

1. its preceding task "Project Manager Decision" has been finished, and
2. the condition of the preceding XOR-gates have the result that the request is further processed, but the budget manager must be involved.

Similar to process instances, task instances get a system-wide unique identifier as well: the *Task Instance ID* (TIID).

In essence, the business layer comprises process and task management components which are necessary to create the correct process and task instances. This is based on the current process, task, and system status. Both, process and task management components use internal mechanisms to keep track of currently running instances which we will describe in detail in Section 2.2.2 below.

Business Object Layer

The third of the three layers is the business object layer. It provides the access to functions of *Business Objects* (BO) or externally located web services. Business objects provide basic functionality of the system on which the process, and especially the task execution is built on. Hence, the business object layer is the operating unit, realizing the executions requested by a user when he is performing a task.

The idea of BOs is to abstract from back-end systems and provide a pre-defined set of functions which can be used by the process engine to execute the tasks of a business process. The BOs and their set of functions are usually specifically defined for one or a set of similar processes. The functions from BOs provide a common interface for operations on, for instance, Enterprise Resource Planning systems (ERP), Customer Relationship Management systems (CRM) or any other data processing units in an enterprise. BOs may even abstract from external service providers which offer, for instance, online services via web service interfaces.

Continuing our previous example on the process "Project Issue Management", assume a purchaser executes the task "Enter Purchase Data" and opens the user interface for entering the data with respect to the additional items which have to be purchased. The interface might display a searchable list of items out of which the user selects items for purchase. Usually, such a list of items is stored in tables on a database located on a particular back-end system. To be displayed, the list must be requested from the database.

On task submission, the set of items the purchaser selected to order, as well as any additional comments he entered must be persisted in the back-end. For both cases BOs provide the necessary functionality. BOs make the respective functions available to be called (possibly as web services), hereby abstracting from the underlying back-end systems. Hence, the BOs take care of all data modifying queries or data selection requests to be performed when a task is executed.

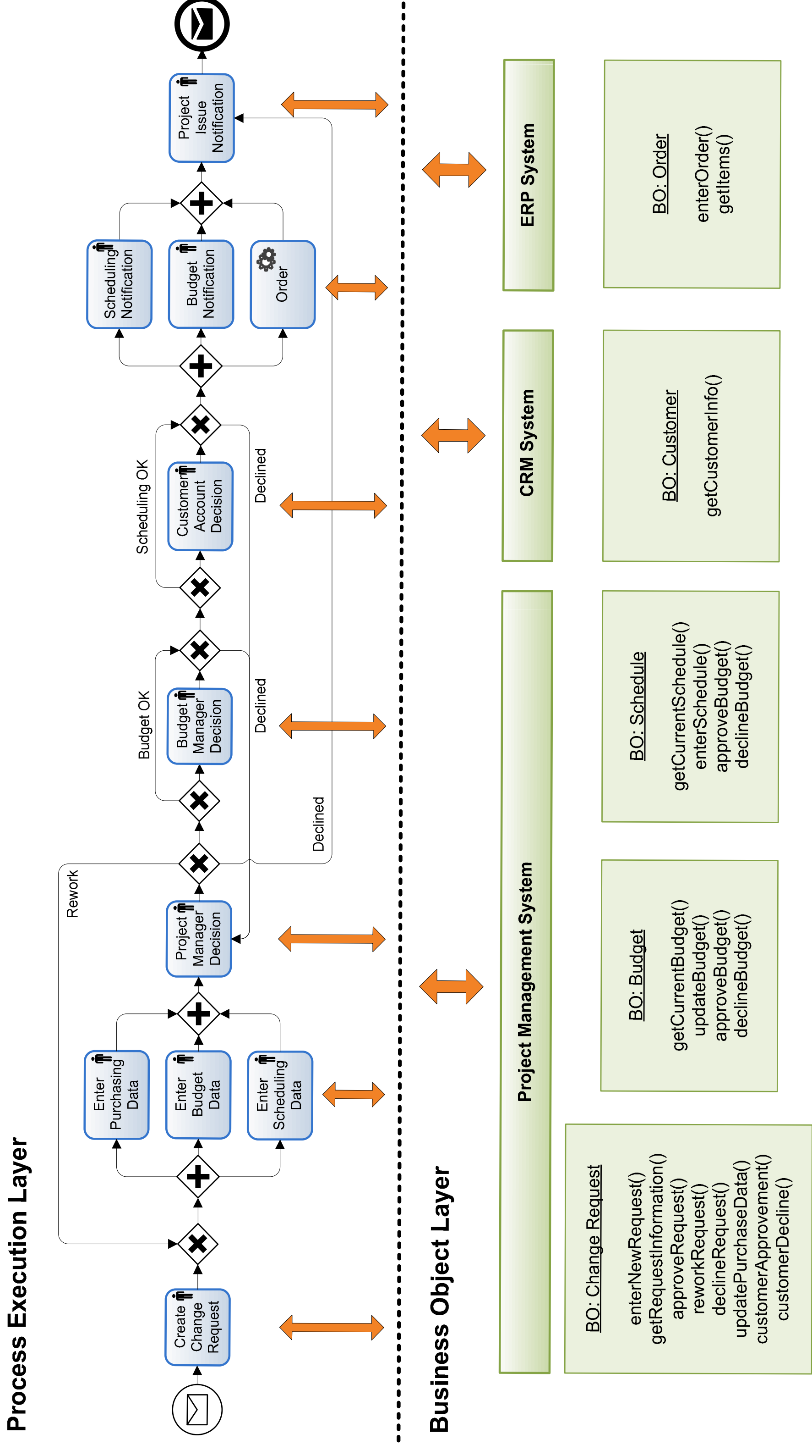


Figure 2-5: Project Issue Management process: Illustration of interaction between Process Execution Layer and Business Object Layer. The Business Object Layer provides access to functionality of back-end systems required for the execution of tasks. The illustration depicts the access to a Project Management System, a Customer Relationship System (CRM), or Enterprise Resource Planning System (ERP) as example.

BOs are provided by service providers realizing operations on different back-end systems or external systems. Figure 2-5 illustrates the interaction between the process "Project Issue Management" given in Section 2.1.2 and the corresponding BOs. The task "Create Change Request", for example, will use the "Project Management System" and its BO "Change Request" to create a new request object on a back-end system.

The task executed by the purchaser ("Enter Purchasing Data") accesses the BO of an "Enterprise Resource Planning" (ERP) system to receive a list of purchasable items. It also accesses the "Project Management System" to update the change request with a list of selected items to purchase.

A last example is the task "Project Manager Decision": it will use the BO's "Budget" and "Schedule" of the "Project Management System" to display the data entered in previous tasks, such that the project manager can make a profound decision. Additionally, it will use the BO "Change Request" to store the approve- or decline-decision when the project manager submits this task.

The business object layer is completely independent from the process execution layer and, hence, does not keep track of which business object calls are related to which process and/or task instance. If standardized interfaces are used, this independence provides the flexibility to exchange systems in the business object layer without the need to change the business process and its tasks as well. The tasks of a process would access the same or similar functionality of a new, exchanged system [70]. Figure 2-5, hence, illustrates one possible set of BOs which might be used to realize the execution of the "Project Issue Management" process.

2.2.2 Life Cycles and System Events

In this section we will illustrate more details about the management of process executions. The process engine is the centre of every process execution. It creates, executes, and deletes instances of processes and tasks. This is done with respect to life cycles which describe the different technical states a process or task instance may adopt. Usually, business process systems define a life cycle for both, process instance and task instance executions, respectively.

The WfMC provides exemplary life cycles in their workflow reference model [72]. The open source workflow engine jBPM from JBoss [62], for example, implements process and task life cycles which are close to the ones provided by the WfMC. Vendors of BPMSs may of course implement their own life cycles.

The life cycles are a major building block for our work presented in this thesis such that we will go into details by describing and defining life cycles in the following sections. For examples and illustrations, we use the life cycles from JBoss jBPM [62].

Process Life Cycle

BPMSs execute process instances according to process life cycles. A life cycle defines a state machine, where at runtime every process instance adopts one of several states. A state reflects the instance's current status. Every state of a process life cycle can be reached through transitions. They transfer the process instance from one state to another. All states and transitions form a directed graph, building the life cycle.

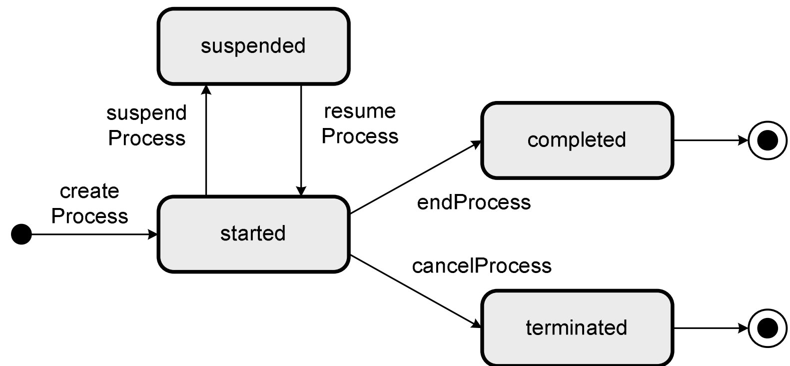


Figure 2-6: Informal illustration of a process life cycle (based on source code of JBoss jBPM [62]).

In case of the JBoss process engine jBPM, the states a process instance may adopt are "started", "suspended", "completed", and "terminated", illustrated with Figure 2-6. After a process instance is initiated, "started" is the first state the instance adopts. The state "suspended" reflects that the respective instance is currently on-hold until a user resumes it. If a process instance is suspended, automatically all active task instances of that process are suspended as well. 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 cancelled outside of the ordinary process execution.

All transitions may be performed during a process execution; an occurrence of a transition is called *Event*. Transitions (and respectively events) may be triggered by user interactions with the BPMS or automatically by the process engine itself. Events which are triggered by user interactions include the creation of a new process instance, as well as suspending, resuming or cancelling it. The remaining event "endProcess" is triggered by the process engine itself. It is triggered as soon as all task instances for the current process instance are finished. In

some cases also "createProcess" is triggered by the process engine. This is the case, if a process is initiated by some of the start events we introduced previously (e.g., timer or conditional start events; see Section 2.1.1).

An event happens during the execution of a process instance. For further processing and reacting on events, each event comes with a set of attributes characterising it. These attributes comprise an *identifier*, a *resource*, a *user*, as well as *instance IDs*, as we will see below.

The identifier reflects the name of the transition (e.g., "createProcess", or "endProcess") upon which the event occurred. Moreover, events happen in the realm of a process instance and can unambiguously be related to the process instance by the instance's PIID. If the event is related to a task instance (executed within a process instance), the event can further unambiguously be related to the task instance by the task instance's TIID.

A process instance is always an executable case of a business process definition. Hence, an event does not only have an identifier such as "createProcess" or "endProcess", but includes the name of the business process definition, i.e., its resource. Events on task instances obviously have the task's identifier as resource; BOs have their identifier entered as resource, respectively.

Informally, assume Alice initiates a new change request referring to the process given with Figure 2-5, a new process instance of the process "Project Issue Management" is created. The event for the process creation would comprise the following attributes:

Event identifier: createProcess
Event resource: Project Issue Management
Event user: Alice
Event instance IDs: PIID(2342)

Life cycles as well as the events of life cycles are important elements for our caching strategy. This requires a more formal definition of life cycles as well as of an event. We will give the definitions next along with several examples illustrating them. We start with the definition of an event.

Definition 1 Event

An *Event e*=event(*n*, *r*, *u*, *I*) is a quadruple, where

- *n* is the identifier of the event,
- *r* is the identifier of the event's resource,
- *u* is the name of the user initiating the event (the name is "SYSTEM" for events triggered by the BPMS), and
- *I* is a set of instance identifiers (i.e., process instance id (PIID) and - if available - task instance id (TIID)).

The set of attributes *n*, *r*, and *u* of the event are called *target*.

We write *e*.n to reference the identifier of an event, *e*.r for the resource, *e*.u for the user, and *e*.I for the set of instance identifiers. Furthermore, referencing an event in general - rather than a specific instance of it - we use its identifier in quotes (e.g., "createProcess", or "assign") as we already used to do throughout this section.

We assume that resource identifiers of an event contain a system-wide unique name, such that process definitions and their tasks can unambiguously be referenced. The resource identifier for a process definition is usually the name of the process such as "Project Issue Management". Tasks can be referenced either by a combination of process and task name, such as "Project Issue Management.Create Change Request", or, if a task name is unambiguous across all process definitions, just by its name "Create Change Request". For our work, we assume that tasks can be uniquely referenced by their task name, such that we use the second, shorter version to reference a task.

In the following we provide examples for events, given the process definition "Project Issue Management" of Section 2.1.2 and the JBoss jBPM life cycles presented in this section.

- Creation of a new process instance, initiated by the user "Alice", and with the PIID "2342" results in the event:

  e_1 = event*("createProcess", "Project Issue Management", "Alice", {*PIID*(2342)})*

- Suspension of a process instance with the PIID "2342", triggered by the user "Alice":

  e_2 = event*("suspendProcess", "Project Issue Management", "Alice", {*PIID*(2342)})*

- Termination of a process instance with the PIID "2342", triggered by the BPMS:

  e_3 = event*("endProcess", "Project Issue Management", "SYSTEM", {*PIID*(2342)})*

Next, we give a generic definition for a life cycle. We model them as deterministic finite state machines. Subsequently, we will give an example of the process life cycle of the JBoss jBPM process engine.

Definition 2 Formal Life Cycle Model

A *formal life cycle model*, represented as final 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 $Q \cap \Sigma = \varnothing$,
- q_0 is the *initial state*, with $q_0 \in Q$,
- δ is the generalized state-transition function: $\delta : Q \times \Sigma \mapsto Q$ describing the possible transitions, and
- F is the set of *final states* with $F \subseteq Q$.

The process life cycle for the JBoss jBPM workflow engine results in a formal life cycle model ($Q, \Sigma, q_0, \delta, F$) depicted in Figure 2-7, where

- $Q = \{s_{inactive}, s_{init}, s_{suspended}, s_{fail}, s_{end}\}$,
- Σ = {createProcess, suspendProcess, resumeProcess, cancelProcess, endProcess},
- $q_0 = s_{inactive}$,
- δ = {($s_{inactive}$, createProcess) $\mapsto$ s_{init}, (s_{init}, suspendProcess) $\mapsto$ $s_{suspended}$, ($s_{suspended}$, resumeProcess) $\mapsto$ s_{init}, (s_{init}, cancelProcess) $\mapsto$ s_{fail}, (s_{init}, endProcess) $\mapsto$ s_{end}}, and
- $F = \{s_{fail}, s_{end}\}$.

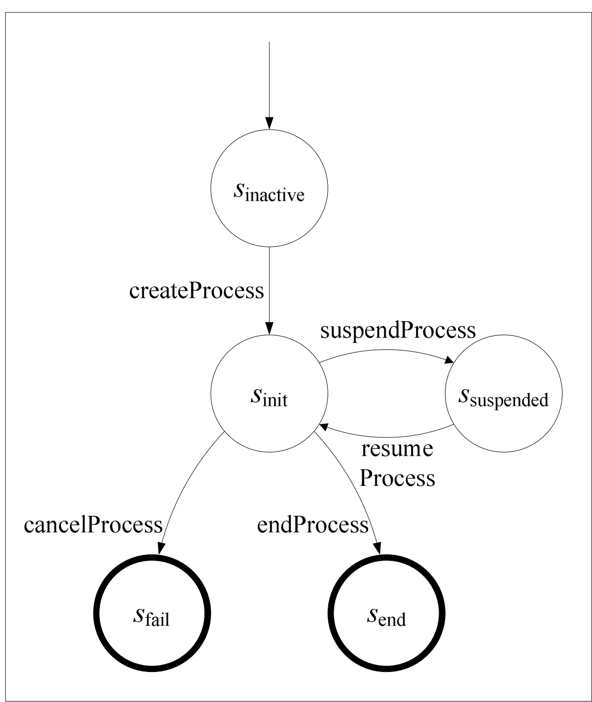


Figure 2-7: Process life cycle of JBoss jBPM as finite state machine

In the next section we will introduce task life cycles.

Task Life Cycle

The task life cycle reflects the states a task instance may adopt. For the JBoss jBPM process engine this comprises the states "created", "started", "suspended", "completed" and "terminated".

The life cycle illustrated as directed graph is given in Figure 2-8. The transitions between the states are similar to the ones of the process life cycle, with the exception of the transitions "(re-)assign" and "startTask". We describe both in the following paragraphs.

The transition "(re-)assign" reflects that a user can be assigned to a task instance after it was created. Furthermore, jBPM allows that an assigned user may be replaced by another user. When re-assigning a new user to a task instance, JBoss jBPM does implicitly revoke the old user before assigning the new one. In the following, we model re-assignments such that the old user is revoked and subsequently the new user is assigned to a task instance. Hence, the re-assignment is a single transition, but comprises the two steps of revocation and assignment, executed by the process engine.

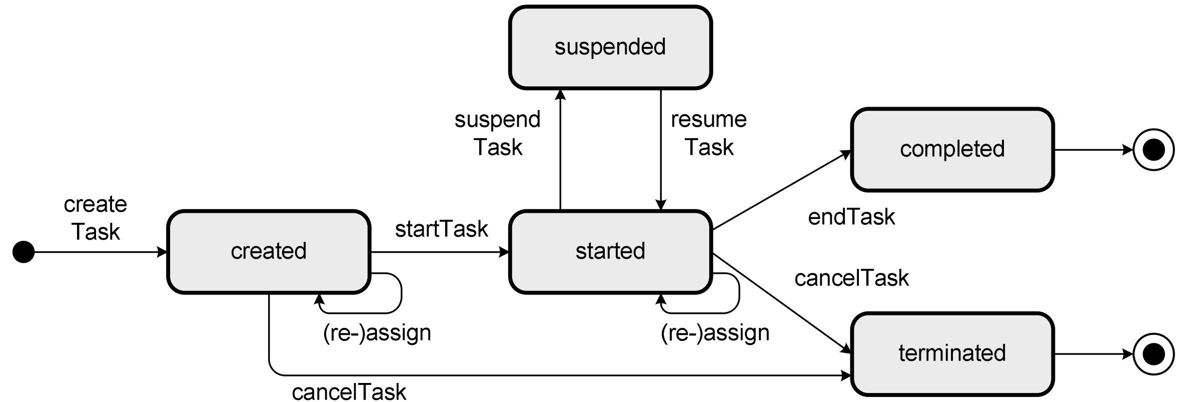


Figure 2-8: Informal illustration of a task life cycle for human-based task execution (based on source code of JBoss jBPM [62], slightly simplified for presentation).

The transition "startTask" reflects that the user opened the task form for the first time.

Similar to the process life cycle, the transitions of a task life cycle reflect *Events* which may occur during a task execution. This also holds for the (re-)assign transitions. Due to our assumption that a re-assignment comprises the two steps of revoking the current user and assigning the new user, in this case the respective transition is reflected by two events.

We gave the definition for events with Definition 1 and provide examples of events based on the task life cycle of JBoss jBPM next. We use the task "Create Change Request" of the process "Project Issue Management" (cf. Figure 2-2) as example.

- Creation of an instance of the task "Create Change Request", where the respective process instance has the id "2342", the created task instance has the id "2266":

  e_4 = event*("createTask", "Create Change Request", "Alice", {*PIID*(2342),* TIID*(2266)})*

- Assignment of a user to an instance of the task "Create Change Request", where the user is "Alice", the respective PIID is "2342", the TIID is "2266":

  e_5 = event*("assign", "Create Change Request", "Alice", {*PIID*(2342),* TIID*(2266)})*

- Re-assignment of a new user "Bob" to the task instance with the TIID being "2266"; recall that we model re-assigning a user as a cycle with the two events "revoke", and "assign":

  e_6 = event*("revoke", "Create Change Request", "Alice", {*PIID*(2342),* TIID*(2266)})*
  e_7 = event*("assign", "Create Change Request", "Bob", {*PIID*(2342),* TIID*(2266)})*

- Finalizing of a task instance by user "Bob", where the respective PIID is "2342", and TIID is "2266":

  e_8 = event*("endTask", "Create Change Request", "Bob", {*PIID*(2342),* TIID*(2266)})*

The process engine from JBoss also supports automated tasks. In fact, automated tasks are the basis on which human-driven tasks are built on. The life cycle for automated tasks is slightly different as the one shown in Figure 2-8. For automated tasks, the process engine would allow that a task instance may be executed without being assigned to a user and without explicitly starting the task's execution. Both, assigning a user and explicitly starting an execution are not necessary for an automated task execution.

In our work, we only concentrate on human centric workflows which require an assignment as well as the starting of an execution. Therefore, we use the task life cycle which includes all elements required for executing human-driven tasks. This is reflected by the illustration given with Figure 2-8.

A state already known from the process life cycle is "suspended". A task instance gets suspended (or resumed) whenever its superior process instance gets suspended (or resumed). Hence, in contrary to a process instance, suspending or resuming a single task instance is not a user's choice, but depends on the process instance in which the task instance is contained.

The task life cycle for the JBoss jBPM process engine results in a formal life cycle model (Q, Σ, q_0, δ, F) which explicitly models a re-assignment of a user as two transitions "revoke", and "assign", such that

- Q = {$s_{inactive}$, s_{init}, s_{start}, $s_{suspended}$, s_{fail}, s_{end}},
- Σ = {createTask, assign, revoke, startTask, suspendTask, resumeTask, cancelTask, endTask},
- q_0 = $s_{inactive}$,
- δ = {($s_{inactive}$, createTask) $\mapsto$ s_{init}, (s_{init}, cancelTask) $\mapsto$ s_{end}, (s_{init}, revoke) $\mapsto$ s_{init}, (s_{start}, revoke) $\mapsto$ s_{start}, (s_{init}, assign) $\mapsto$ s_{init}, (s_{start}, assign) $\mapsto$ s_{start}, (s_{init}, startTask) $\mapsto$ s_{start}, (s_{start}, suspendTask) $\mapsto$ $s_{suspended}$, ($s_{suspended}$, resumeTask) $\mapsto$ s_{start}, (s_{start}, cancelTask) $\mapsto$ s_{fail}, (s_{start}, endTask) $\mapsto$ s_{end}}, and
- F = {s_{fail}, s_{end}}.

The respective finite state machine is depicted with Figure 2-9.

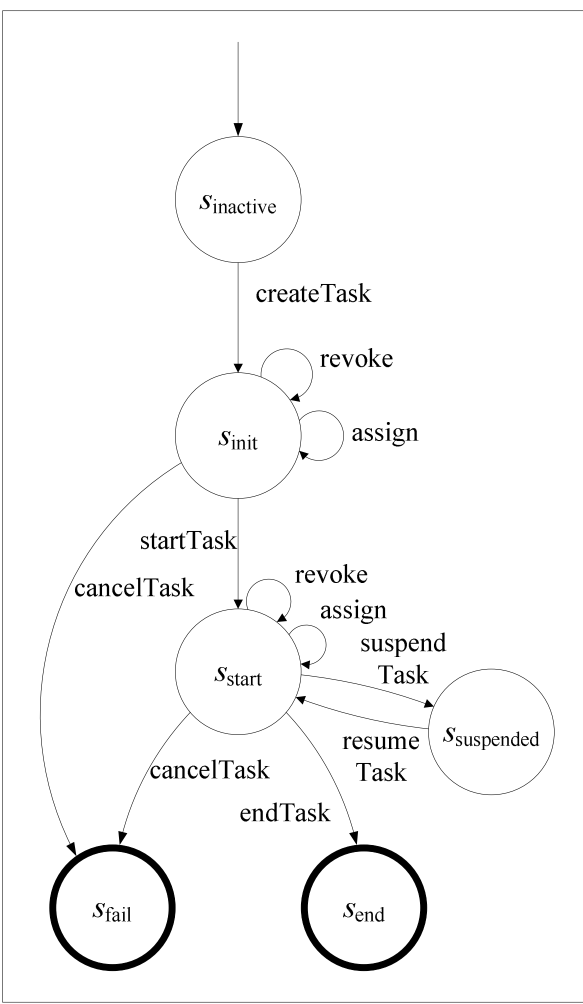


Figure 2-9: Task life cycle of JBoss jBPM as finite state machine

In the following section, we illustrate and define events based on the interaction with business objects.

Business Object Events

We previously described that task executions involve business objects (BO). During a task execution, the functionality of those BOs are called, necessary to execute and realize a task's objective on a respective back-end or other, external systems. A BO may offer the information a user requires to perform the task (e.g., status information about a project, purchase order, or a particular system), it may offer the functionality to add, update, modify, or delete data in a back-end system, or it even might call web services provided by external service provides.

Each call of a functionality provided by business object (in short: BO-call) is considered as *Event*. In the following, we will give a few examples of events for BO-calls, based on the exemplary process "Project Issue Management" and the BO relations illustrated with Figure 2-5.

- A project manager Alice enters input for a new change request and finishes the first task. The data Alice entered is stored as a new change request object in the back-end. The BO "Change Request" abstracts the respective creation of a new change request object by providing the function "enterNewRequest". When Alice submits her input, the BPMS calls this function to create a new object and to complete the task. We assume the process instance has the PIID "2342", and the task instance the TIID "2266". The related event for the BO-call is:

  e_9 = event*("enterNewRequest", "Change Request", "Alice", {*PIID*(2342),* TIID*(2266)})*

- Alice performs the task "Project Manager Decision". There are three BO-calls necessary to execute this task. The first and second calls are executed to retrieve information about the current budget and schedule from the back-end system. This information should be displayed to the Alice whenever she opens the task's form to execute the task. The third one is the submission of the manager's approval decision. We assume the process instance has the PIID "2342", and the task instance the TIID "2270". The three respective events are:

  e_{10} = event*("getCurrentBudget", "Budget", "Alice", {*PIID*(2342),* TIID*(2270)})*
  e_{11} = event*("getCurrentSchedule", "Schedule", "Alice", {*PIID*(2342),* TIID*(2270)})*
  e_{12} = event*("approveRequest", "Change Request", "Alice", {*PIID*(2342),* TIID*(2270)})*

2.3 Summary

In this chapter we provided an overview about business process management, business process definitions, as well as a respective modelling notation BPMN which we will use throughout this thesis for the illustration of processes.

A first example of a business process called "Project Issue Management" illustrated the use of business processes on a real-world example. We will use this process as running example for further process relevant illustrations in the following chapters.

We further gave an introduction into business process execution. The illustration of the three-layered architecture as well as life cycles and events build cornerstones for all following chapters where we introduce the details about dynamic access control for business process-driven environments (Chapter 3), the caching strategy (Chapter 5), the caching heuristic (Chapter 6), and respective performance analysis (Chapter 8).

3 Access Control for Business Process-driven Environments

Large enterprise systems increasingly contain a large amount of confidential data (e.g., financial data, customer data) which are administrated and processed by IT supported business processes. Regulatory requirements such as Basel II [2], Sarbanes Oxley [21, 53], as well as data protection laws impose access control restrictions on such systems.

Consequently, modern business process-driven systems, supporting applications such as Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), or Supply Chain Management (SCM), more and more require fine-grained, complex, and dynamic access control enforcement.

In this chapter we will provide a brief introduction into access control enforcement for business process-driven systems. It is important to understand the requirements the dynamic nature of business process systems impose on enforcing access control evaluations such that restrictions on caching strategies for access control can be considered. We will describe the requirements on access control and present the criteria to build a dynamic strategy for caching and storing access control decisions.

Furthermore, the application of access control for business process executions requires an architecture supporting the evaluation and enforcement of access control policies. Within this chapter, we will introduce an access control reference architecture specifically developed for business process-driven systems. The reference architecture is taken from ORKA [46], which is the context in which our work was created. The reference architecture will function as basis for our work and will be extended with respect to our caching strategy in later chapters. Our extensions, however, are generic enough such that we assume most of today's business process-driven systems are able to adopt our caching strategy.

The structure of this chapter is divided into two main parts. In the first part we present a set of important access control constraints tailored to business process executions. For each constraint we show the criteria to be considered for caching strategies. In the second part we introduce the reference architecture for the evaluation and enforcement of access control decisions in business process-driven systems.

3.1 Security Policy: Permissions & Constraints

One part of access control enforcement is to specify the criteria based on which access should be granted or denied. A user (or application respectively), states an

access control request against a system which has to be evaluated. Each access control evaluation results in an *access control decision*, which is based on a set of defined *permissions* and *constraints*. The evaluation is performed by a security evaluation component (usually called *Policy Decision Point* (PDP)) which computes whether access should be granted or denied.

Permissions generally specify a set of access rights, which allow (or deny) a user to interact with a system. Permissions are often specified using a role-based access control model [50]. Constraints, on the other hand, restrict these access rights, such that access is only granted if further conditions are fulfilled. There are static and dynamic constraints.

We will introduce role-based access control, as well as both types of constraints in the following sections, and we will further discuss specific constraints relevant for business process execution.

3.1.1 Role-based Access Control

Today's enterprise systems usually use role-based access control (RBAC) which is a widely known access control model [5, 9, 18, 22, 25, 32, 49, 50, 61, 64, 67] and ANSI[3] standard [1]. The standard introduces different role-based access control models. The first model, *Core RBAC*, defines four entities: users (U), roles (R), permissions (P), and sessions (S). In accordance with [1], we briefly introduce all four entities next.

Roles are a semantic construct, which describe a job function within an organization [49]. Recall our example process "Project Issue Management" we introduced in Section 2.1.2. The process illustration (cf. Figure 2-2) already shows that different roles are involved during process execution: Project Manager, Purchaser, Controller, Scheduler, etc.

A role contains all **permissions** required to perform the duties the specific job function of the role reflects. In the context of our work, permissions describe the authorized interactions a user may perform on processes, tasks, or business objects. Hence, the role "Project Manager", for instance, contains the permissions to perform the three tasks a project manager is responsible for, namely: "Create Change Request", "Project Manager Decision", and "Project Issue Notification".

Users may be assigned to one or more roles. Users reflect human beings which may authenticate themselves with a (workflow) system by their user name, a unique identifier. In the role-based access control model, a user obtains the permissions of the roles of which he is member of.

[3] ANSI: American National Standard Institute, http://www.ansi.org

Sessions reflect a semantic construct which lets a user activate a subset of the roles he is member of. Assume a user *u* is member of a set of roles *r*; during a session, the user only receives the permission obtained through a subset of roles $r' \subseteq r$ he actually activates. This equates a session to an abstract user *u'* which is assigned to the set of roles *r'*. Hence, according to this model, every user *u* may be impersonated by one of the abstract users *u', …, u*, where the abstract user is only member of the set of roles *r', …, r* respectively.

For the execution of a business process instance, we assume that each instance is considered as session and a user only uses at most one of the abstract users for the execution of one instance.

Role-based access control is often assumed to be more scalable than direct user-permission assignments [63]. The assumption is the set of permissions a user requires for his work changes more often over time than the permissions required for a certain job junction. Hence, permissions assigned to a role are more stable than the assignments of users to a certain role [49].

Roles may be structured in a hierarchy. Roles on a higher level of the hierarchy subsume the permissions of their sub-roles. In the ANSI standard [1], this is the second model, called *Hierarchical RBAC* [1].

In the third model, *Constrained RBAC,* constraints are introduced. These constraints state administrational restrictions on user-role-assignments and role-permission-assignments. A prominent example is the constraint type for mutual disjoint roles. It requires that a user may only be member of at most one of the set of these roles. The ANSI standard describes this constraint as *Static Separation of Duties*. Further constraint types described by Sandhu et. al [50] are *Cardinality*, and *Prerequisite Roles*. Cardinality constraints define a maximum number of members of a role. Constraints for prerequisite roles define that a user may only be assigned to a role A if the user is already assigned to role B.

The above mentioned constraints apply for administrational changes of the security policy and do not require any information of a currently running workflow system. We consider them as *static constraints* and will consequently refer to these constraints as S*tatic Separation of Duties* (SSoD), S*tatic Cardinality*, and *Static Prerequisite Roles,* respectively.

Permissions in RBAC allow defining access rights for processes, tasks, and business objects and are independent of process or task instances. RBAC only defines that a user may, for example, claim the task "Project Manager Decision", if the user is member of the role "Manager". Hence, only using RBAC would give a user access to all *instances* of the task "Project Manager Decision" currently active in the system.

Thomas and Sandhu already state in [65] that workflow systems require access control models which allow dynamic access rights based on the current system status. Instead of giving a user access to all instances of a task, the user should, for example, only be able to perform task instances of a process he actually started. It might be important to define that a project manager may only perform the task "Project Manager Decision" for those change requests he is actually responsible for. This cannot be expressed by static constraints and requires constraints which take *dynamic context information* into account.

These constraints include *Dynamic Separation of Duties* (DSoD), *Binding of Duties* (BoD), *Dynamic Cardinality Constraints*, and general *Attribute-based Constraints*. We will briefly introduce them in the following sections.

3.1.2 Dynamic Separation of Duties

Business process-driven systems require that tasks in a process may be defined to be performed by different users (e.g., due to law regulations such as Sarbanes Oxley [53] or Basel II [2]). Such restrictions are called *Dynamic Separation of Duties* (DSoD) constraints [5, 9, 10, 18, 25, 28, 32, 49, 54, 61].

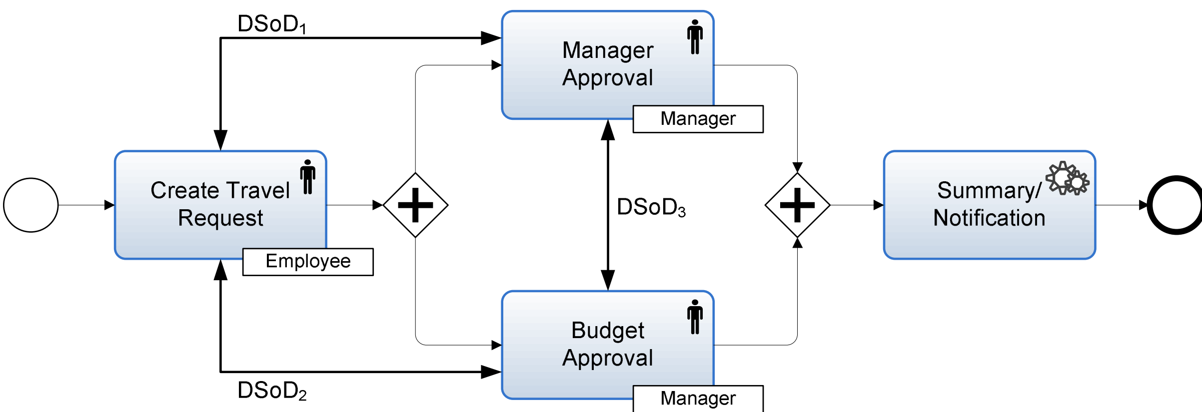


Figure 3-1: Travel request process annotated with Dynamic Separation of Duties (DSoD) constraints

In workflow systems, DSoD constraints separate the execution of tasks between multiple users. More commonly, this is known as four-eyes-principle and established to prevent users from either intentionally misusing access to multiple tasks for their own benefit, or to prevent users for unintentionally making mistakes by having multiple persons involved. Tasks which fall under a separation of duties constraint are called *exclusive tasks*.

Recall our introductory process "Travel Request". As an example, we want to ensure that the task "Manager Approval" may only be performed by a user which is member of the role "Manager". We also require that no user is allowed to approve his own travel request, and

that at least two managers have to have a look at each request, hence, a different manager must approve the budget. For illustration, we informally annotated the travel process in Figure 3-1 with three DSoD constraints $DSoD_1$, $DSoD_2$, and $DSoD_3$, where each of them specifies two exclusive tasks which may not be performed by the same user. Generally, a DSoD may be expressed as follows.

DSoD: Task *t* and *t'* may only performed by different users.

A separation of duties constraint is per se not limited to two tasks. The list of exclusive tasks can contain multiple tasks. A more general form of DSoD is required for use-cases in [54, 55] which state that a user may be allowed to perform a maximum number of tasks out of a set of exclusive tasks. Hence, the above mentioned example is a specific case where the number of exclusive tasks is two, and the user may perform a maximum number of one task out of the two. The general version of a DSoD constraint may be expressed as follows.

DSoD: Given a set *E* of exclusive tasks, a user may perform a maximum number of *x* tasks out of E, where $x < |E|$ and $|E|$ is the size of E.

The access control evaluation of DSoD constraints requires context information. In fact, it requires the information which tasks a certain user already claimed or performed within the same process instance. Only with the list of tasks a user already claimed or executed it is decidable, whether access to another exclusive task may be granted or must be denied.

The context information about the tasks a user performed on a specific process instance is available by the BPMS. Hence, the BPMS acts as a *context provider*, delivering the information the PDP requires for the evaluation of DSoD constraints.

3.1.3 Binding of Duties

In contrary to DSoD, there might be tasks within a business process which have to be performed by the same user. This type of constraint is called *Binding of Duties* (BoD) [10, 61]. The constraint specifies a set of bounded tasks. The user performing one of them has to perform the complete set of bounded tasks.

In our example process "Project Issue Management" there are three tasks which should always be performed by the same user, given one process instance. For instance, if a project manager performed the task "Create Change Request", the same user should also perform the task "Project Manager Decision", and also receive the "Project Issue Notification" at the end of the process (cf. Figure 3-2).

Figure 3-2: Binding of Duties example

Hence, as soon as a user claimed "Create Change Request", it should be the same person who performs "Project Manager Decision", and "Project Issue Notification".

A more general formulation for BoD is as follows.

BoD: Let *E* be a set of tasks. A user performing task $t \in E$ must also perform all other tasks $t' \in E$.

Similar to DSoD, access control evaluations of BoD constraints require context information about the user who is obliged to perform the set of bounded tasks. The user performing one of the tasks performs them all; this is determined during runtime of a workflow execution. The information which user performed one of the tasks is available by the BPMS. Hence, the BPMS acts again as context provider, delivering the information the PDP requires for its evaluation of BoD constraints.

3.1.4 Dynamic Cardinality Constraints

During the execution of one process instance, one of the process's tasks may be instantiated with multiple instances. Cardinality constraints define that a user may only perform a defined number of instances of the same task within one process instance [5, 61].

When executing a process, there are at least two possibilities that a task is instantiated more than once. The first possibility is that a process definition contains one or more loops. Tasks within the loop may be instantiated and executed multiple times, depending on how often a process instance iterates through the loop(s). An example is the task "Project Manager Decision" within our example process "Project Issue Management". This task might be instantiated as long either the project manager decides that the change request should be reworked, or the budget manager or customer declines the proposed changes, also resulting in a rework of the request. In both cases the change request returns to the three specialists for revising the request and consequently returns on the project manager's desk with another instance of the task "Project Manager Decision".

The second possibility for multiple instances of one task is that a task is specifically defined to be executed having multiple instances of the same task executed in parallel. This is a special type of task which allows that not only one instance of a task is created, but multiple

ones. Each of the generated instances must be performed by a user. The number of instances created during process execution is defined within the process definition.

The cardinality constraint defines the number of such task instances a user may perform at most.

Dynamic Cardinality: A task *t* of the business process *p* may at most be executed *n* times by the same user within one process instance.

For the evaluation of such a constraint, the PDP requires the number of task instances a user already claimed or performed within a given process instance. If the number raises the allowed threshold, the user is declined to claim any further instances of the respective task.

The number of tasks a user already performed is available from the BPMS and dependents on a user's interactions with the system by claiming and executing task instances.

3.1.5 Attribute-based Constraints

Security often requires that access is only granted if selected attributes fulfil pre-defined conditions [9, 22, 25, 61, 63]. A possible requirement may state that a task must only be executed during working hours. Another example is the requirement that a respective process or task instance must be in an error state before an administrator is allowed to act on it.

Attribute-based constraints comprise one or more conditions which are checked during access control evaluations. For the constraint that a task may only be executed during working hours, the evaluation of the constraint must - informally - evaluate whether, for instance, the conditions "6 a.m. ≤ Now( )" and "Now( ) ≤ 8 p.m." hold, where "Now( )" is a *dynamic attribute value* returning the current system time.

A more general expression for an attribute-based constraint is as follows.

Attribute-based: A permission restricted by an attribute-based constraint may only be granted if the set of conditions *C* defined by the constraint is satisfied.

Dynamic attribute values may be provided by various sources. These include the BPMS itself as well as context information from external systems. As example for an attribute value from an external system, Schaad et al. state in [54] the scoring value about a potential customer for a bank as being a critical value to decide whether a higher ranked manager must be involved in the process.

3.2 Caching Access Control Decisions

Access control decisions are the result of the evaluation of access control requests based on a specified security policy. Our work introduces a strategy for caching access control decisions. This includes that access control decisions are stored at a specific location for future reference. The set of cached decisions can be queried with respect to current access control requests instead of performing a regular access control evaluation.

Caching access control decisions requires that cached decisions are valid as long as they are stored in the cache. An access control decision is considered *valid* if, and only if the cached result is the same decision a regular access control evaluation component would return at the point in time the cache is queried.

Cached decisions, however, may become invalid over time. There are two reasons. The first reason is that the security policy (i.e., the specified permissions and constraints) changes, or in case a role-based access control model is used, the user is removed from a role such that the respective permission is no longer part of the user's access rights. In our work we consider the security policy to be static, meaning it remains unchanged. (A discussion about policy changes with respect to our work, however, can be found in Section "Discussion on Updating Cache Entries" in Chapter 9.)

The second reason for cached decisions becoming invalid is changing context information. Access control decisions are based on the evaluation of permissions. Permissions may be further restricted by additional evaluations of *dynamic* constraints. Dynamic constraints require context information for their evaluation; hence, access control decisions depend on such context information. Context information may change over time and may consequently render stored access control decisions based on this information invalid.

Any context information which changes over time is considered as *dynamic context information.* A subset, however, is considered as *change-detectable context information.* General dynamic context information may change continuously or in an arbitrarily fashion, change-detectable context information only changes at specific, detectable events, broadcasted by the workflow system.

The current system time, for instance, is an example for dynamic context information changing continuously. An attribute-based constraint such as the one presented in the previous section (i.e., access is only granted during working hours) requires the current time for its evaluation.

Further, any context information from external locations (e.g., web services) may be considered as information which is not under the influence of the local application and, hence, may change arbitrarily without further notification. An example is the previously

mentioned scoring value about a potential customer of a bank, provided by an external rating service.

We consequently define constraints which rely on dynamic context information as *dynamic context constraints*. The above mentioned constraints DSoD, BoD, Dynamic Cardinality, as well as Attribute-based Constraints belong into that category.

Definition 3 Dynamic Context Constraints

Dynamic context constraints are access control *c*onstraints which require dynamic context information for their evaluation.

DSoD, BoD, and Dynamic Cardinality of the above constraints, however, require context information which also belongs into the subcategory of change-detectable context information. We consider DSoD as example. Recall our process "Travel Request" which we annotated with DSoD constraints (see Figure 3-1 oben) to ensure that no user is allowed to approve his own travel request, and that two different users must approve the request and corresponding budget.

From a business process execution perspective, which user actually performs a task, is defined during runtime when a user claims a task. At the moment a user actually claims one of the exclusive tasks defined with a DSoD constraint, the same user becomes restricted to claim any of the other exclusive tasks. Hence, the information whether a user already claimed one of the exclusive tasks is relevant whether the same user may be allowed to claim any of the other exclusive tasks. This information, however, does not alter in an arbitrary way, but only at the time a user claims an exclusive task.

Consequently, from a cache management perspective, the event of a user claiming one of the exclusive tasks of a DSoD constraint signalises, that previously cached decisions for all other exclusive tasks of the same constraint may be invalid and must be updated.

We consider the list of tasks a user claimed and executed as context information which is necessary for the evaluation, whether a user may claim another exclusive task. With the occurrence of an event of a user eventually claiming an exclusive task, this context information changes and renders access control decisions relying on the context information possibly invalid.

The signals for such context changes are the "assign"-events for an exclusive task, broadcasted by the workflow system. The same holds for the above mentioned constraints BoD and Dynamic Cardinality which also rely on the context information about tasks a user already claimed.

Constraints relying on change-detectable context information are called change detectable context constraints.

Definition 4 Change-detectable Context Constraints

Change-detectable context constraints are dynamic context constraints which rely on dynamic context information where changes can be detected by broadcasted events.

In contrary to change-detectable context constraints, the result of an attribute-based constraint evaluation is only valid at the point in time of its evaluation. Hence, the check whether the constraint's conditions hold must remain *open* until the point at which the access control decision is actually needed. We call them *open constraints* (OC).

Definition 5 Open Constraints

Open constraints are dynamic context constraints which rely on dynamic context information where changes cannot be detected by broadcasted events.

In essence, there are constraints which rely on dynamic context information. We call them dynamic context constraints. The set of dynamic context constraints is dived into two sub-sets: change-detectable context constraints and open constraints. The mapping of a constraint to one of the sub-sets depends on the context information required for the constraint's evaluation.

3.3 Reference Architecture for Access Control in Process-driven Systems

The work discussed in this thesis was carried out in the context of the German-funded project ORKA (Organisational Control Architecture) [46]. One of the main goals of ORKA was to develop a framework which allows dynamic access control enforcement for modern business process-driven systems. One major outcome important for our work is a generic reference architecture for realizing access control enforcement for workflow systems. The architecture takes all three layers of process execution into account.

The generic system architecture for business process-driven systems is depicted with Figure 3-4 and comprises the three layers distinguishing user interactions, business process management, and functional access to back-end systems via business objects or web services we already introduced in Chapter 2.

Although the three presented layers might be tightly linked with each other, from an access control perspective, it is still the case that each layer can be accessed independently. For example, methods of business objects can also be called without the necessity that the call has to be mediated by a process engine; the functionality exposed by the process engine can

also be used by third party software delivering their own user interface adapted for their own needs. For this reason it is crucial that each of the presented layers incorporates its own access control enforcement components realizing enforcement on each layer.

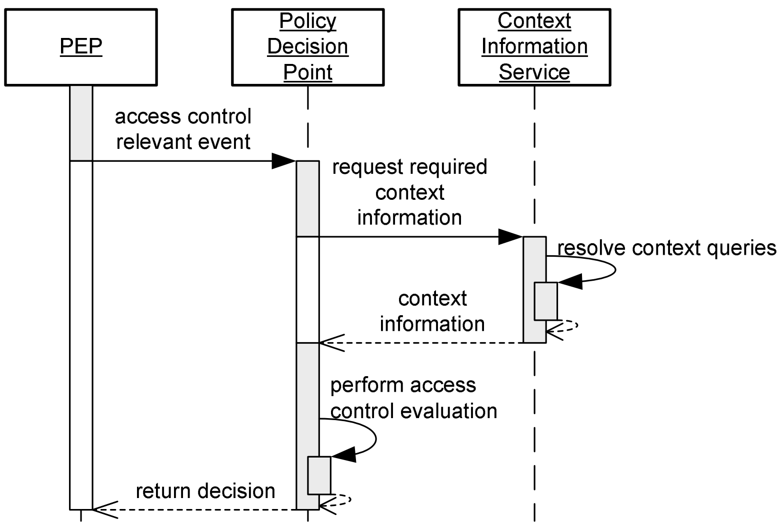


Figure 3-3: Sequence diagram for an access control evaluation using the request-response paradigm [46]

In large enterprise systems access control is usually based on the request-response paradigm (see Figure 3-3 for illustration) [15, 44, 46, 68]. It comprises the interaction between a *Policy Enforcement Point* (PEP) and a *Policy Decision Point* (PDP). A PEP receives system events which should be checked against the system's security policy. We call such events *Access Control relevant Events*. The PEP sends an access control request including the information about the received event to the PDP. The PDP is the component which evaluates the access control request based on the system's security policy. After evaluation, the PDP returns an access control decision back to the PEP. The decision contains the required access control statement for the PEP to either block the further execution of the access control relevant event or to grant its execution.

Definition 6 Access Control relevant Event
An *Access Control relevant Event* is an event which must be checked against the system's security policy.

Each of the three layers contains one or more PEPs to enforce the access control decisions of the PDP (see Figure 3-4 unterhalb for illustration). The PDP itself is connected to a *Policy Storage* and a *Context Information Service.* The policy storage contains and manages a system's security policy. The security policy comprises the rules according to which the PDP decides whether an access request should be granted or denied. The context information

service acts as gateway for additional information the PDP requires for its access control evaluations.

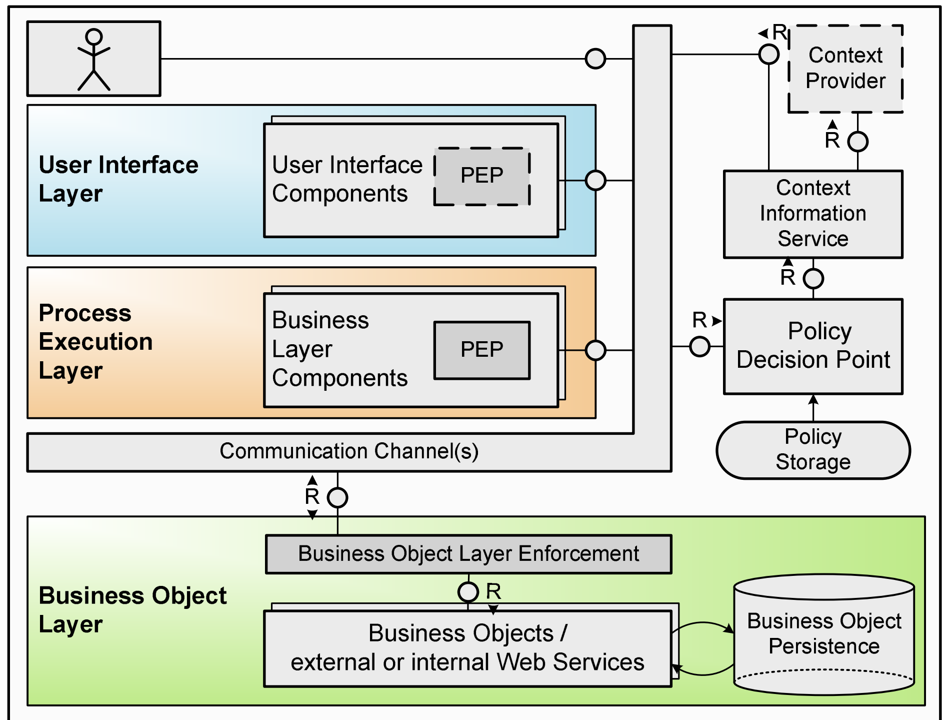


Figure 3-4: Reference Architecture for Business Process-driven Systems [46]

We showed in the previous section that the evaluation of access control constraints often requires dynamic context information (e.g., about the current status of the BPMS, or about the user requesting access). This context information may come from any component within a BPMS itself, or from different *Context Provider* which are located internally within the same enterprise or may be hosted externally. The context information service mediates access to these context providers.

In the following paragraphs we go through each layer, describing the respective access control enforcement in more detail.

Access control on user interface layer comprises access control enforcement for user interface and control elements such that the user only sees those elements he is actually able to use. Examples for such elements within a BPMS might be buttons to create new processes, to cancel, suspend, or resume running processes, to claim a task, to delegate tasks to other users, or further elements which allow a user to interact and manage the system.

In most cases, access control on user interface layer serves a usability requirement. All of the above mentioned user interactions are functions provided by the business layer and its

business process management. Their actual execution will be performed on the business layer and in accordance with the assumption that the three layers are separately accessible, again checked against the system's security before finally performed. Hence, access control on user interface layer serves the requirement that the user is aware, which interactions he is actually allowed to perform and which not. If a user cannot cancel one of the running process instances, for instance, the interface should not even provide him with a perceived ability that he is allowed to cancel it, but should show the cancelation as disabled or not display the respective cancel button at all.

The same holds of course for the general worklist (GWL). Recall our description in Section 2.2.1, that the user should only see those task instances within the system he is actually able to claim.

Access control on the process execution layer comprises controlling which events for process and task management a user triggers are legit requests to be executed. In dynamic environments access control must be checked and enforced at the very moments a user actually requests the execution of an event. Access control on the user interface level is not sufficient: for instance, between the time the user views a task within the GWL and the time he actually claims it, the access control evaluation of dynamic constraints might already result in a different access decision.

On the business layer, process and task management occurs. All access control relevant events a user might trigger with respect to process or task instances are depicted as transitions on the process and task life cycles. Which events of the process and task management are access control relevant is assumed to be defined with the security policy of a BPMS. Without loss of generality, we assume for our work that only events which require user interaction are checked against the security policy. For the process life cycles of JBoss jBPM, for example, the following events can be triggered by a user: "createProcess", "cancelProcess", "suspendProcess", and "resumeProcess". For the respective task life cycle, events which may require user interactions are "assign", "startTask", and "cancelTask".

Access control on the business object layer comprises access enforcement for exposed functions of business objects (BO). Recall, BOs provide functions which are required for task executions. Functions of BOs which are meant to be invoked for human task executions, should only be invoked by users which are currently assigned to a respective task. Each task often calls one or more functions of a BO. ORKA showed that it is important that security policies define dependencies across layers. In particular, access control checks for BO functions are required for checking whether the user calling the function is actually assigned to a respective task. This means, if a function of a business object is called, it is not only checked whether the user is generally allowed to call this function, but also whether the user is actually assigned to a task, which requires the call of this particular function.

3.4 Summary

In this chapter we presented a reference architecture for access control within business process-driven environments. The reference architecture was developed in the German-funded project ORKA [46]. We will use and extend it as basis for our caching strategy in the following chapter.

We also presented that the evaluation of access control requests may include the evaluation of one or more dynamic context constraints which require dynamic context information. This context information, however, may change over time, possibly rendering already evaluated and cached decisions invalid. Stored access control decisions may not only base on one, but on different access control constraints. If the context information for just one of these constraints changes, the complete stored decision may become invalid.

Moreover, we introduced dynamic context constraints. Dynamic context constraints reflect constraints which require dynamic context information for their evaluation. One subset of these constraints is considered to be change-detectable, which allow our cache management to react on altering context information; for the remaining subset (i.e., open constraints), context changes may not easily be detected such that other means for caching will be introduced.

4 Related Work

In this chapter we present related work. We first give an overview in work about caching and prefetching of information. Afterwards we focus on the area of access control and introduce further work on caching, as well as other methods of approaches to optimize performance for access control evaluations.

4.1 Caching & Prefetching

Caching is a widely known and used functionality to improve performance or availability of today's computing systems. Caching strategies are either application independent [36], or specifically tailored and applied for various system and application domains. Caching, for instance, is used for data storage and databases [11], file systems [47], web servers [17], processors [33], memory access [40], as well as application domains such as data compression [4], or access control [6, 15, 29-31, 68, 69].

In most cases, a cache component stores results which are anticipated to be required during at a later point in time, and it is assumed that retrieving them from the cache is much faster than calculating the same result a second time. In some cases, caches are also used to provide increased availability. These types of caches store last known results, such that they are available in case their original source is not available. Google's web cache [23] is a prominent example. In the area of access control, Crampton et. al introduced a caching strategy which puts focus on availability as well [15, 68]; we describe the details of their approach below.

Our caching strategy (introduced Chapter 5) is tailored to business process-driven environments. In particular, it addresses caching in the domain of access control and relies on prefetching access control decisions. Prefetching is a well known principle for caching in various areas. Examples are web applications [17], optimizing I/O disk management [47], as well as improving memory accesses and processors [33, 40]. There are different approaches to determine which information should actually be pre-fetched.

Prefetching information always requires knowledge based on which algorithms or rules, which information should be pre-fetched and actually be stored in the cache. Most of the time, this knowledge is application specific and requires deep knowledge of the underlying requirements the cache should fulfil.

Often, statistical or historical information about previous events are consulted. In web applications, for instance, the goal is to prefetch hyperlinks and their embedded images. In [17], Duchamp uses (past) user actions to determine highly clicked references of a given

web page. This information is distributed to other users such that references, most likely to be consumed by the user, can already be fetched in background.

Another information source for realizing prefetching are applications themselves disclosing hints by indicating that, for instance, a particular file is going to be read sequentially four times in succession [47]. The knowledge which information will be needed next (i.e., which file will be called and how often will it be read) is disclosed via the programmer by revealing knowledge of the application's behaviour. Another example for this approach is shown by Mowry [40]. He uses knowledge available during compile time to optimize the memory subsystem for microprocessors. Prefetch instructions are directly inserted into the code while its compilation.

In our work, prefetching is used to selectively pre-evaluate those access control requests anticipated to be needed during the further execution of a process instance. Our pre-evaluations are based on the knowledge of the underlying process models (i.e., system life cycles, and business process definitions), relations between tasks and business objects, as well as the respective security policy. By analysing these information sources, we are able to generate rules, allowing a very effective anticipation of future access control requests.

In the area of access control, pre-computing access control decisions was to our knowledge firstly proposed by Beznosov [6]. His work suggests to speculatively pre-evaluate access control requests whenever resources and bandwidth allows the additional effort. The pre-computation should be done by guessing which decisions might be required in near future, based on the history of prior access control requests. In particular, Beznosov proposes the idea of a publish-subscribe architecture where the enforcement and decision components are decoupled and a policy enforcement point (PEP) actively subscribes to one or more policy decision points (PDP). This allows that speculatively pre-computed access control decisions (called "junk authorizations") to be directly broadcasted to their respective subscribers, i.e. PEPs.

Beznosov already recognizes that for some "history-sensitive policies, dynamic separation of duties, or other types of policies which require subject-rights to be consumable" additional methods are required, such that access control decisions may be cacheable. The reason is that access control decisions based on context sensitive policies might become invalid as the context changes. This requires additional effort such that the cache remains in a state at which all its entries remain valid, i.e., the cache returns the same result as the PDP would directly answer, given an access control request.

Business process environments require the implementation of such context sensitive security policies. Our caching strategy addresses this and provides the possibility to specify specific rules based on which our cache management is capable to update cached access control decisions in such a way the cache remains in a valid state.

4.2 Caching Access Control Decisions

Improving the performance of access control evaluations in a distributed environment is a well known research topic. There is work which particularly uses caching as proposed solution. Examples are work from Borders et al. [8], Crampton et al. [15], Spencer et. al [58], and Wei et al. [68] which we will present in the following paragraphs.

Spencer et. al present **FLASK [58]**, a security framework with caching capabilities for operating systems. The framework's goal is to provide a security architecture which supports a wide range of security policies for managing access control rights within a given system. The framework claims to provide the flexibility required for the propagation of access rights for single objects of the system. Caching is used such that access control decisions can directly be stored at the component regularly requesting access for objects. The challenge Spencer et. al identify for their system are the propagation of policy updates, providing the solution that caches register themselves to be notified if the security policy changes such that the caches can react and update their content accordingly.

Although the authors acknowledge that there are also context sensitive security policies which may require dynamic context information for policy evaluations, their framework does not support updating cached decisions with respect to changing context information.

Borders et. al [8] also describe policy enforcement as integral part of applications which more and more require fine-grained access control. The authors introduce **CPOL**, a C++ framework for policy evaluation, using caching of access control decisions to improve the performance of access control responses. The access control evaluation engine of CPOL returns *access tokens* encapsulating a set of allowed rights. Access rights are given based on the evaluation of rules which further may contain conditions; conditions must evaluate to *true* for the rule to apply. These conditions are Boolean expressions and can take input from the system's environment. According to the authors, this is especially important when enforcing access control for location-aware services. Hence, conditions may define expressions such that access may only be granted during a given time frame or if the user is located at pre-defined locations.

CPOL uses caching for performance improvements. Cache entries are created whenever a new access request is evaluated. A cache entry stores the access token, a *CacheCondition*, as well as information about the last system context used to obtain access. The cache only returns the access token if the respective cache entry is still considered valid; otherwise a regular access evaluation is performed. Whether a cache entry is still valid is determined by analyzing the CacheCondition. The CacheCondition is an object which is implemented by an application developer. It is used by the authors to specify a cache entry's invalidation criteria which state timeouts and movement tolerances. When creating the cache entry, the timeout

and movement tolerance is determined and stored according to an expected invalidation of the accompanied access token. On request of a cache entry, a function "StillGood" implemented by the CacheCondition-object checks whether the current system is still within the given tolerance. This, of course, requires that the cache receives the current system state (e.g., the current time, or a user's current location as illustrated by the authors) based on which such validation checks can be executed.

Hence, Borders et. al deal with the problem of dynamically changing context information by providing an invalidation mechanism for their cached access control decisions. We are going to use a similar approach for handling those access control decisions which are based on open constraints; our approach, however, is not limited to determine whether an access control decision is still valid, but allows actually to evaluate whether access should be granted without further contacting the PDP. Moreover, for change-detectable context constraints, our caching strategy even provides specific update capabilities such that regular access control evaluations as well as additional queries for fetching additional context information can be avoided.

Crampton et. al propose a secondary and approximate authorization model (**SAAM**) [15]. This is the closest related work to our caching approach, and we compare our work against SAAM in Chapter 8. The authors introduce a general model for caching access control decisions while proposing a method allowing to infer new access control decisions by using the information of earlier access control requests and decisions stored with the cache. The authors distinguish between the notions of *primary* vs. *secondary*, as well as *precise* vs. *approximate* access control decisions. Primary decisions are answers directly evaluated by the PDP, while secondary responses are delivered by a caching component. Precise access decisions are either primary decisions from the PDP or primary decisions from the cache which have previously been stored. Approximate access control decisions are results which are inferred based on the set of earlier requests and decisions.

Crampton et. al describe the purpose for secondary and approximate caching being twofold. The first reason stated is that current fault tolerance techniques fail to scale for large populations while remaining cost-effective. As solution, Crampton et. al provide the just described model allowing for approximating decisions also if the connection to the PDP is interrupted. The second reason is "the high cost of requesting, making, and delivering an authorization due to communication delays".

Crampton et. al applied SAAM for Bell-LaPadula policies [3], called $\textbf{SAAM}_{\textbf{BLP}}$ [15]; Wei et. al introduced $\textbf{SAAM}_{\textbf{RBAC}}$ [68] for role-based access control policies. $SAAM_{BLP}$ and $SAAM_{RBAC}$ are both based on the same principle; we will go into details for the latter as it is directly related to our work.

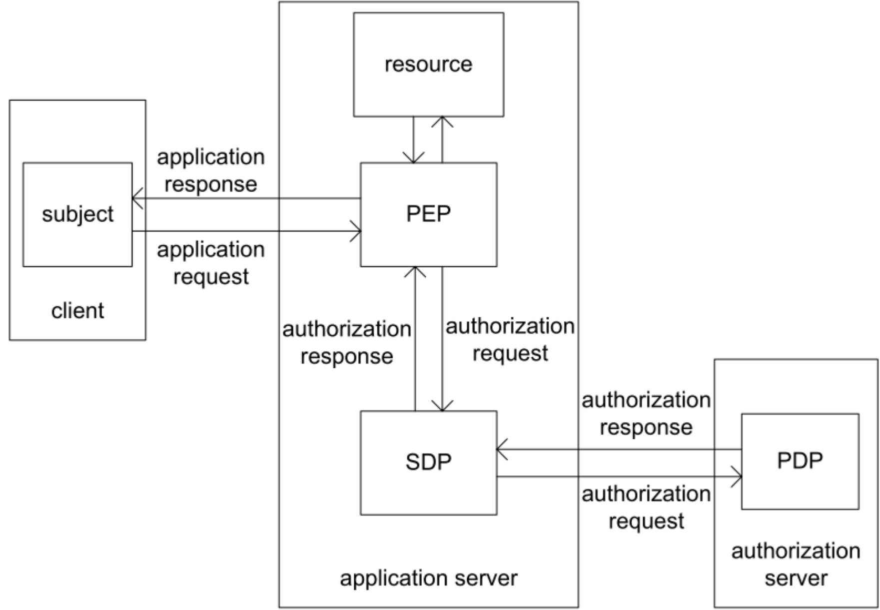


Figure 4-1: SAAM builds on the request-response paradigm and adds a secondary decisions point (SDP) [68]

$SAAM_{RBAC}$ builds upon the previously described request-response paradigm (cf. Chapter 3) and adds a secondary decision point (SDP) as shown in Figure 4-1. The SDP functions as a cache with the extended ability to not only answer access control requests with precise access decisions, but also by answering them with computed approximate decisions - based on already cached access control decisions.

The authors develop different algorithms to create and update the cache each time there are new pairs of access control requests and decisions available; they further provide algorithms to approximate access control decisions for responses to access control requests.

For the approximation of access control decisions the authors assume that the security policy is defined according to the RBAC standard. It is further assumed that each access control request contains the set of roles a user is assigned to. For creating the cache, the algorithms try to reconstruct which permissions the different roles have, based on the knowledge gained by monitoring the primary access control responses which the PDP returns, given an access control request. In particular, the cache is structured in two lists: $Cache^+$ and $Cache^-$.

For an access control request which has been granted, $Cache^+$ stores the corresponding roles which led to the granted authorization (i.e., resulting in a mapping of an access control request to a set of roles). The roles are taken from the request, as it is assumed that each request contains all roles a user possesses. A user usually possesses more roles than

required to receive a granted authorization; hence, $Cache^+$ stores an over-approximation of the roles leading to a granted authorization.

Respectively, for each access control request which has been denied, $Cache^-$ stores the corresponding set of roles which led to the denial. Deny responses are further used to correct the over-approximation of $Cache^+$-entries. For a denied access control request p containing a user's set of roles R^-, none of the roles $r \in R^-$ possesses the required permission. Hence, all roles which led to a denial (i.e., R^-) are removed from the set of roles R^+ for the same request p stored in $Cache^+$.

New secondary access control decisions are inferred from the entries in $Cache^+$ and $Cache^-$. In principle, if the roles contained within an access control request fully match the set of roles stored for an already cached request response within $Cache^+$, the request is granted; respectively, if the roles of the request are part of a set of roles stored for an entry with $Cache^-$, the request is denied.

$SAAM_{RBAC}$ requires that role-based security policies are used; in particular, dynamic access control constraints such as presented in Section 3.1 are not supported. For RBAC-policies, the authors tested their caching strategy, showing that 30%-128% of access control requests can be answered with approximate responses compared to standard caching, which only returns precise decisions stored in the cache.

$SAAM_{RBAC}$ will be one candidate of caching strategies against which we will compare the caching strategy we present in our work. Details for the comparison, the respective results and an extended discussion can be found in Chapter 8.

4.3 Performance Improvements within the Area of Access Control

Caching access control decisions is only one way of increasing a system's performance. There are other ways to improve the performance of access control evaluations. In this section we introduce further work which deals with performance improvements in the area of access control.

The **IBM Tivoli Access Manager** [27] uses replication of a central policy database to deal with performance and reliability improvements. The Access Manager is a policy-based access control system for IT-system landscapes. The Access Manager's *Authorization Service* is a central entity, providing the respective access authorizations for clients. It comprises an Authorization Policy database, a Management Server, and an Authorization Server. The Authorization Policy database contains the systems security policy, the Management Server maintains the policy database, and the Authorization Server determines a client's right to access a requested resource. For performance and availability reasons, the policy database

(and other parts of the authorization service) may be replicated for each application. Policy changes are "pushed" from a master Policy database to its replicas.

Liu et. al describe a policy evaluation engine **XEngine** [34], specifically developed for the access control policy specification language XACML [44]. XACML is an XML-based, application independent policy specification language, developed and standardized by OASIS [43], a not-for-profit consortium for open IT-standards.

XACML allows specifying security policies which consist in a hierarchical structure of elements, namely policy sets, policies, rules, and conditions. In particular, a policy set may contain a list of policies or policy sets. A policy contains a set of rules while each of them may define one or more conditions. Each of these elements - with exception of conditions - may define a *target* upon which it is decided whether the respective element and its sub-elements apply for an access control request. A target comprises three optional identifiers for user, resource, and action. An access control request is always queried for a specific user, claiming access on a resource by using a particular action. A policy set, a policy, or a rule is considered for evaluation if the user, resource, and action elements of an access control request match the ones specified by the respective targets. Eventually, if the target of a rule matches the elements of an access request, the rule's conditions are evaluated; if the rule's target matches and the conditions are fulfilled, the rule's effect (i.e., *permit* or *deny*) is returned, otherwise *not applicable* is returned.

The authors of XEngine claim that current policy evaluation engines for XACML such as Sun's reference implementation [60] perform a "brute force searching by comparing a request with all rules in an XACML policy" [34]. The authors present three ways for an optimized evaluation. Firstly, all string values are converted into numerical values. Matching a target with an access request as described above, usually requires string comparison. By replacing all strings within targets with numerical values, the required time for comparisons gets improved. Secondly, the authors reduce the hierarchical complexity and transform the "numericalized" policy into a "flat structure", having only one conflict resolution method "first-applicable". Thirdly, the already transformed policy is converted into a tree structure for an "efficient processing [of] requests".

Their experiments show that XACML files of different sizes and structures may be processed up to four times as fast as when using Sun's reference implementation.

There is further work dealing with transforming XACML policies to increase the performance of access evaluations. **Marouf** et. al [35], for instance, reorganizes the XACML policy according to the incoming access control request. In fact, they categorize access requests by the user of each request; first, to find the policy applicable and, second, to find an *execution vector* which states the order in which the rules of the policy are applicable. Previous access

requests are used to weight policies and rules based on their evaluation-frequency and their complexity. Evaluation-frequency thereby refers to the number of times a policy or rule gets evaluated during a given time interval, complexity refers to the number of operations required to evaluate a rule.

Both weights are used as "cost-values", leading to a re-ordering of the XACML policy such that an execution sequence for evaluating the rules of the policy with minimal cost can be found, and hence, only a minimum number of rule evaluations are required to reach a decision. While transforming the policy, it must preserve the policy's original specification, which access control requests should be granted and which should be denied.

Their work was also compared with the Sun's PDP implementation, showing a performance increase by orders of magnitude. The same holds for work which was done by **Miseldine** [39], which introduces an algorithm to optimize an XACML policy with respect to improve the performance of access control evaluations, but also having in mind that the resulting, optimized policy should be consumable without modifications to a respective PDP.

All of the above mentioned work increases the performance by means of optimizing the access to a security policy and its evaluation. These approaches, however, still require regular access control evaluations for each access control request sent to the PDP. This includes fetching dynamic context information from different context providers such that a security policy can be evaluated based on real-time context data.

We focus on caching access control decisions. This still requires the evaluation of access control requests by the PDP where above mentioned structural optimizations of the security policy itself may be applied; the majority of performance improvement will, however, come from the fact that access control requests can directly be answered from the cache. We analyse and present a comparison between different caching strategies as well as scenarios where no caching is implemented in Chapter 8.

4.4 Summary

In this section we introduced related work. We gave an overview about caching and pre-fetching in general, followed by the introduction of work which especially deals with caching in the area of access control. Finally, we introduced further work which optimizes access control evaluations by other means than caching; namely, replication and transformation of security policies.

5 ProActive Caching Strategy

The goal of ProActive Caching presented in this thesis is to decrease the overall response time experienced by the user when he 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 contribute to this overall goal, we concentrate on performance gains by reducing the response time attributed to access control evaluations. The response time can be reduced if a significant step - the access control evaluation itself - can be shortened or totally removed.

This chapter describes the general ProActive Caching strategy. We first give an overview about the idea of ProActive Caching. Afterwards, we introduce the strategy itself, as well as the underlying caching heuristic and additional components required within our reference architecture (introduced in Section 3.3). We will further define all necessary aspects of *Cache Entries* and the *Caching Heuristic*. Finally, we go into details showing how to deal with access control decisions relying on dynamic context information, which usually makes them infeasible to be cached with regular caching strategies. ProActive Caching allows caching of such dynamic constraints as well.

5.1 Overview

Many executions of business processes require that certain user interactions with the system are restricted by access control such that only pre-defined authorized users are allowed to perform them. In terms of process execution, related work usually postulates that a task in its whole should be restricted in its execution [5, 14]. Whereas in most cases this is a valid abstraction, for our caching approach we further have to consider that the execution of a task usually happens on different layers across a business process management system (BPMS). This means we must consider multiple access control checks on different layers for the execution of one task.

The evaluation of access control requests is time consuming, especially if dynamic context constraints have to be considered. These constraints require that external context information has to be fetched and analysed such that the PDP is able to reach an access control decision. Recall the small example about the general worklist (GWL) in Section 2.2.1: we assumed that 300 process instances require about 100 access checks to display a user's GWL, resulting in at least two seconds delay - solely originating from calculating the access control decisions. Depending on a system's architecture and the PDP's physical location with respect to the location of the workflow system, the delay might even increase due to, e.g., network delay.

For improving the overall experienced performance of the system, we propose a caching strategy for reducing the response time needed for access control evaluations. The strategy aims at caching access control decisions such that access control requests are directly be answered from the cache rather than regularly evaluated by the PDP. The idea is that getting an access control decision from the cache is by magnitudes faster than waiting for a standard, regular request evaluation.

Current caching strategies [8, 15, 27, 68] lack the possibility to cache access control decisions based on *dynamic context constraints* (cf. Definition 3, Section 3.2). The evaluation of dynamic context constraints depends on the current system state, i.e., the PDP is not stateless. Hence, an access request evaluated by the PDP uses the currently available context information. The same access control request evaluation might have a different result, if the context information changes between the two requests. The problem with cached decisions based on dynamic context constraints is that they may become invalid over time.

A second limitation is that existing caching strategies usually do not provide answers from the cache for a first time an access control request appears. This leads to *cache misses,* as for the access control request the respective access control decision is not contained in the cache, yet, but has to be evaluated regularly by the PDP. Only after at least one regular evaluation, the resulting decision is stored with the cache. In consequence, the same request must at least appear twice such that it can be directly answered by the cache and, hence, benefit from caching.

The objective of the caching strategy presented in our work is to provide a solution that optimizes the availability of cache entries by anticipating which access control request evaluations are required during the execution of a business process. This allows us to pre-evaluate access control requests and store the respective access control decisions such that they are available when needed. This enables us to even provide answers for first-time queries. Furthermore, our solution enables caching of access control decisions based on dynamic context information as well. We describe our overall caching strategy in the following section.

5.2 Strategy

The general idea of ProActive Caching is to use "knowledge" in the system to anticipate the access control decisions required during execution of a business process instance. Furthermore, the same "knowledge" is used to remove cached decisions which are obsolete.

Access decisions are pre-computed based on the anticipation of their necessity and cached prior to the point in time they are actually needed. This means they are already available

when the process execution comes to the point the respective access control request should regularly be evaluated. If all access requests required during the execution of a business process can be anticipated and pre-evaluated, the cache contains all access decisions required for the execution of exactly those business process instances currently active in the system. Hence, all access control requests can directly be answered by the cache with minimal delay and without waiting for a regular request evaluation by the PDP.

The "knowledge" used for the anticipation of required access control decisions comprises several *information sources*, such as the process and task life cycles, the business process definitions, or the relations between tasks and business objects.

Next, we start with introducing the prerequisites on the environment and business process execution on which our strategy is built on. Afterwards, we present the essence of our caching strategy and provide general information about the caching architecture required for the strategy. The information sources will be introduced in a dedicated Section 5.3 following the current one.

5.2.1 Environmental Prerequisites

Our caching strategy prerequisites two aspects of business process-driven systems: the request-response paradigm for access control enforcement and the possibility to anticipate upcoming events based on currently occurring events.

The **first prerequisite**, the request-response paradigm, describes the interaction between a policy enforcement point (PEP) and a policy decision point (PDP). We described the underlying reference architecture in Section 3.3. In large enterprises and environments supporting the service-oriented architecture (SoA), a system's PEPs and the PDP are usually separated. The PEPs are part of the application, requesting access decisions from the PDP. The PDP evaluates the request and returns an access control decision as response, which again is enforced by the PEPs. This means, there are dedicated access requests for which specific access decisions are returned as answers.

The PDP is an autonomous component which reacts on receiving access control requests for their evaluation. This especially allows that not only the business process application may send access requests to the PDP, but also allows a cache management component to send requests. In consequence, a cache management component may send access control requests on its own discretion, such that the resulting decisions can be stored with a cache, independent of the rest of the system.

The mentioned request-response paradigm further allows that a cache component can intercept access control requests originally meant to be sent to the PDP. The cache answers

the request, given it contains the respective cached decisions; otherwise, it forwards the request to its original receiver, the PDP, for a regular access evaluation.

The **second prerequisite** is the ability to anticipate upcoming events. The execution of business processes relies on business process definitions, the system life cycles and its events. Process and task management of any business processes execution follows according to the life cycles defined with the system (cf. Section 2.2.2). A business process execution results in a series of defined events reflecting that a process or task instance transits from one state into another state. It is important to note that the events do not occur in an arbitrary order, but always according to the life cycles pre-defined paths along the transitions between the states a process or task instance can adopt.

Every process execution starts with the creation of a process instance and ends with the termination of this instance. During its execution, the process and task life cycles define which events may happen, and in which order they may happen. For instance, given the process life cycle of JBoss jBPM (see Figure 2-7) it is clear that if the event "createProcess" occurred, the instance adopts the state "started". The next events are either "suspendProcess", "cancelProcess", or "endProcess" as those are the only three transitions leaving the state "started".

After a process instance is created, the instances for the tasks are created and executed. Which tasks are performed and their order of execution is defined with the process definition. Similar to process instance executions, each task instance execution follows in accordance to the task life cycle (cf. Figure 2-9).

Figure 5-1 shows one possible order of events for an execution of the "Project Issue Management" process (introduced in Section 2.1.2). The figure illustrates events on the x-axis and the different states the process and task instances can adopt on the y-axis. The graph shows a small fraction of the first events occurring during the execution of the complete process. We go through the graph next.

The process execution starts with the instantiation of a process instance, illustrated with the start of the red line at the state "started" in the lower half of the graph. The line's position according to the y-axis indicates the current state of the process instance.

The second event "createTask" depicts that a first task instance is created, illustrated with the first blue line starting at the state "created" in the upper half of the graph. The assignment of a user to the task generates the next event "assign" on the x-axis. The state of the task instance changes to "started", as soon as the user begins working on the task by opening the task's user interface.

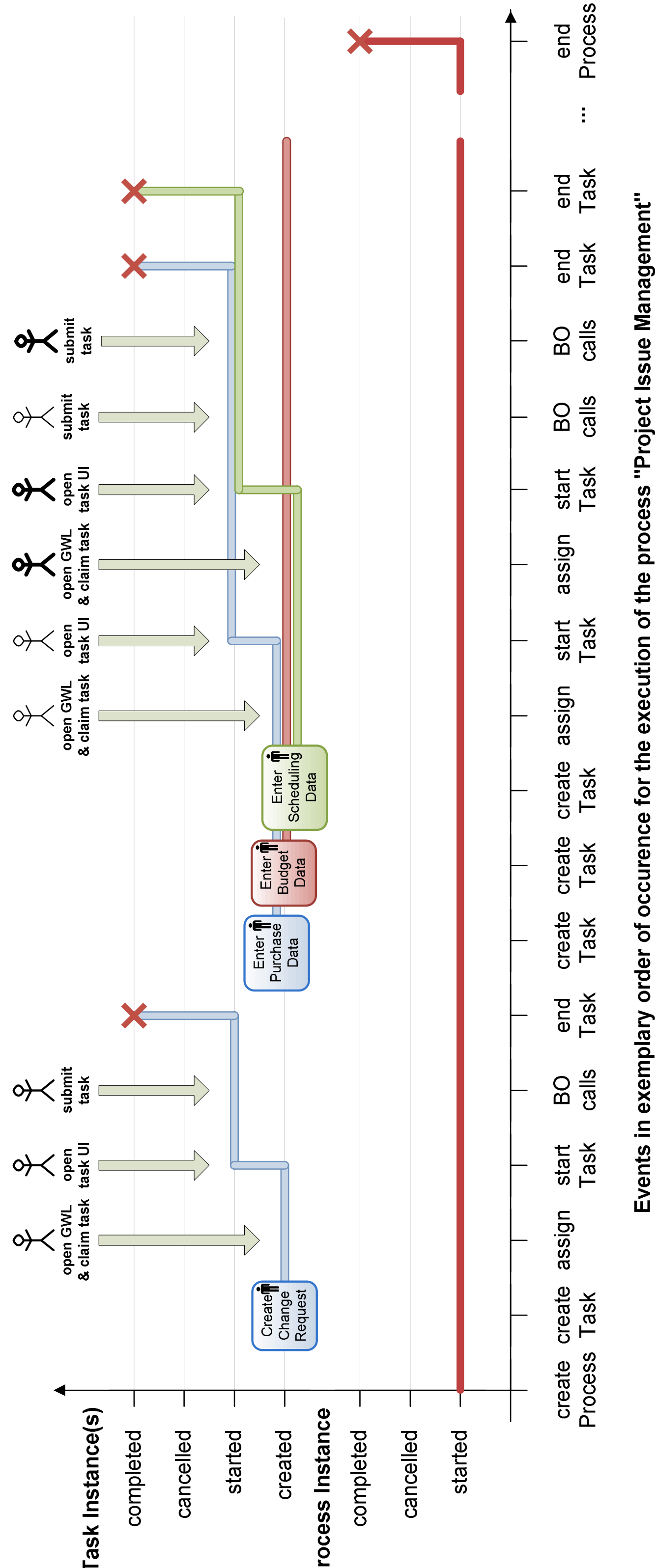


Figure 5-1: Exemplary order of events for the "Project Issue Management" process, given the JBoss jBPM life cycles

After the user entered the data and submits it, the BOs are called to store the user's input in the back-end, depicted as "BO-calls". With the event "endTask", the task instance transits into the state "completed" after all BO-calls have successfully been finished. Now, task instances for the three parallel upcoming tasks are created, claimed and executed; similar as just described for the first task. This happens with all tasks until the process is finished and its instance transits into the state completed.

The example shows, that the overall order of events is dependent on the structure of a business process and the interactions with the users. Within one instance, however, the order of events is predictive. The set of possible events succeeding a given event is clearly defined with the system's life cycles; possible succeeding events are the set of events leaving the state an instance currently adopts. The execution of task instances within jBPM, for example, will in most cases follow the sequence of "createTask", "assign", "startTask", events for "BO-calls", and finally "endTask"; variations, of course, are possible according to the task life cycle. This execution of process and task instances according to their life cycles allows us to anticipate required access control requests during a business process execution as it is always known, which events may follow - given previous events that led the current execution to its current state.

In summary, the two prerequisites for our caching strategy are the use of a request-response paradigm for access control checks, and the ability to anticipate upcoming access control relevant events, based on pre-defined system life cycles, events, and business process definitions.

5.2.2 Caching Strategy and Environment

Events reflect status changes on the current system status, broadcasted by the BPMS. They allow a cache management component to react on the events occurring during a process execution.

Our caching strategy (illustrated in Figure 5-2) defines that a cache management component triggers the pre-evaluation of access control requests required for those events, which are (1) access control relevant events, and (2) expected to be one of the upcoming events during the execution of a business process instance. Pre-evaluations of access control requests are triggered based on currently broadcasted events. The evaluations themselves are performed by the PDP; its answers (i.e., access control decisions) are stored in the cache.

Moreover, based on the broadcasted events, the cache management also recognizes that a process or task instance is finished. Given such events, the management component revokes out-of-date cached decisions no longer needed.

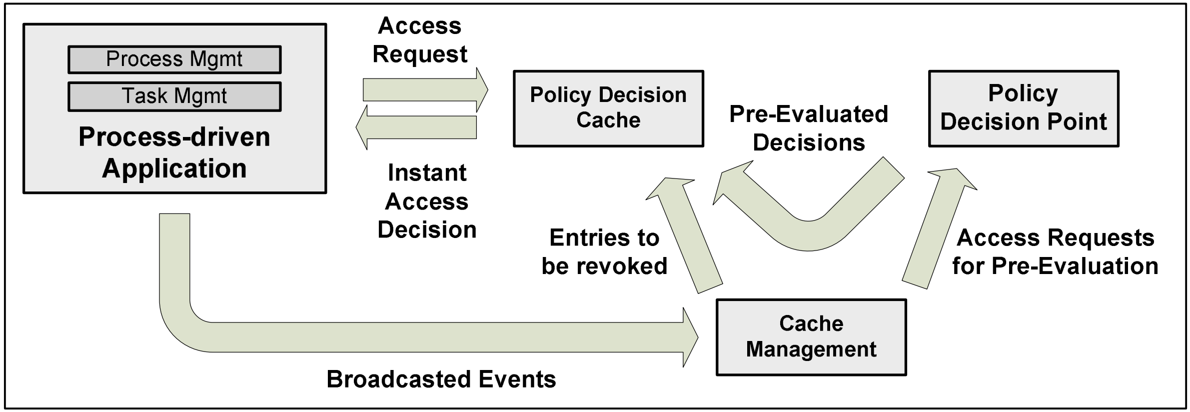


Figure 5-2: Caching Strategy Overview

The whole cache management is based on a caching heuristic we will introduce in Section 5.3 below. It defines the events at which access control requests should be pre-evaluated and at which events cached decisions can be revoked.

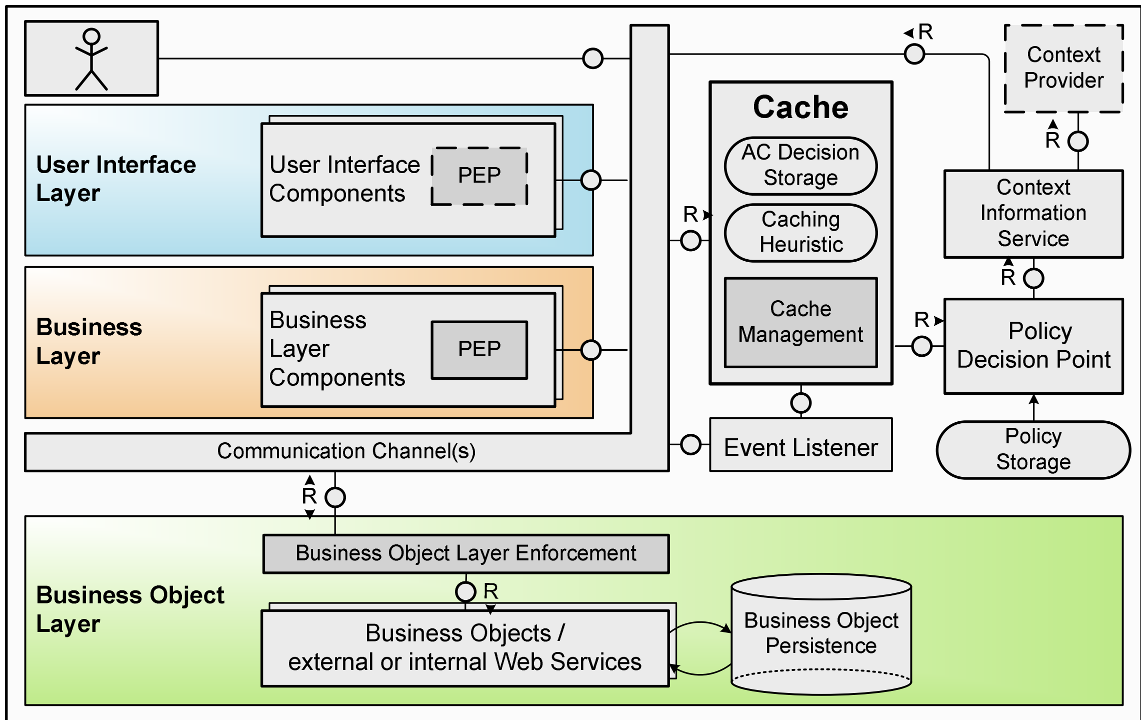


Figure 5-3: Caching Architecture

Our caching strategy requires an extension of the previously presented reference architecture, adding a *Cache* and an *Event Listener*. In Figure 5-3 the cache is directly placed between the business process application (represented by the three layers and their communication channel) and the PDP. The cache itself includes a storage for cache entries

(named *Access Control (AC) Decision Storage*), a storage for *Caching Heuristics*, and the *Cache Management* component.

Furthermore, the extended architecture depicts an *Event Listener* which is responsible for receiving the information, which life cycle or business object events occurred during the execution of a business process. These are forwarded to the cache and its cache management component, such that the cache management can react appropriately according to the current status of the system.

The cache contains pre-evaluated access control decisions, called *Cache Entries*. A cache entry comprises all information reflecting for which access control requests it was pre-evaluated. This includes an identifier of the access control relevant event, the respective names of the resource and user, as well as the respective process instance identifier. It also contains the access control decision (e.g., "PERMIT" or "DENY").

We define a cache entry as follows and provide an example thereafter.

Definition 7 Cache Entry

A *Cache Entry* is a quintuple *ce* = CE(*e, r, u, piid, d*), where

- *e* is an event identifier,
- *r* is a name of a resource,
- *u* is a name of a user,
- *piid* is a process instance identifier, and
- *d* is an access control decision.

We write *ce*.e to reference the name of an event, *ce*.r for the resource, *ce*.u for the user, *ce*.piid for the process instance identifier, and *ce*.d for the access control decision. Similar to events, we assume that the resource identifier should contain a system-wide unique name such that process definitions and their tasks can unambiguously be referenced.

Assume that an access control decision for Alice was pre-evaluated and stored with the cache. The cache entry states that Alice is allowed to cancel a respective process instance for the process "Project Issue Management" with PIID *2342*. The corresponding cache entry is given next.

ce_1 = CE("*cancelProcess*", "*Project Issue Management*", "*Alice*", PIID(*2342*), "*PERMIT*")

The cache can be queried with access control requests. A request is compared against all entries the cache contains. An entry matches if the elements *e, r, u,* and *piid* of the entry are equal with the elements of the access control request. If the cache contains a matching cache entry *ce*, the decision *ce*.d is returned; if it does not contain a matching entry, the

cache forwards the request to the PDP for a regular evaluation. There are no two entries in the cache having identical matching elements *e, r, u,* and *piid*.

A sequence diagram of the actual access control check during runtime is depicted with Figure 5-4.

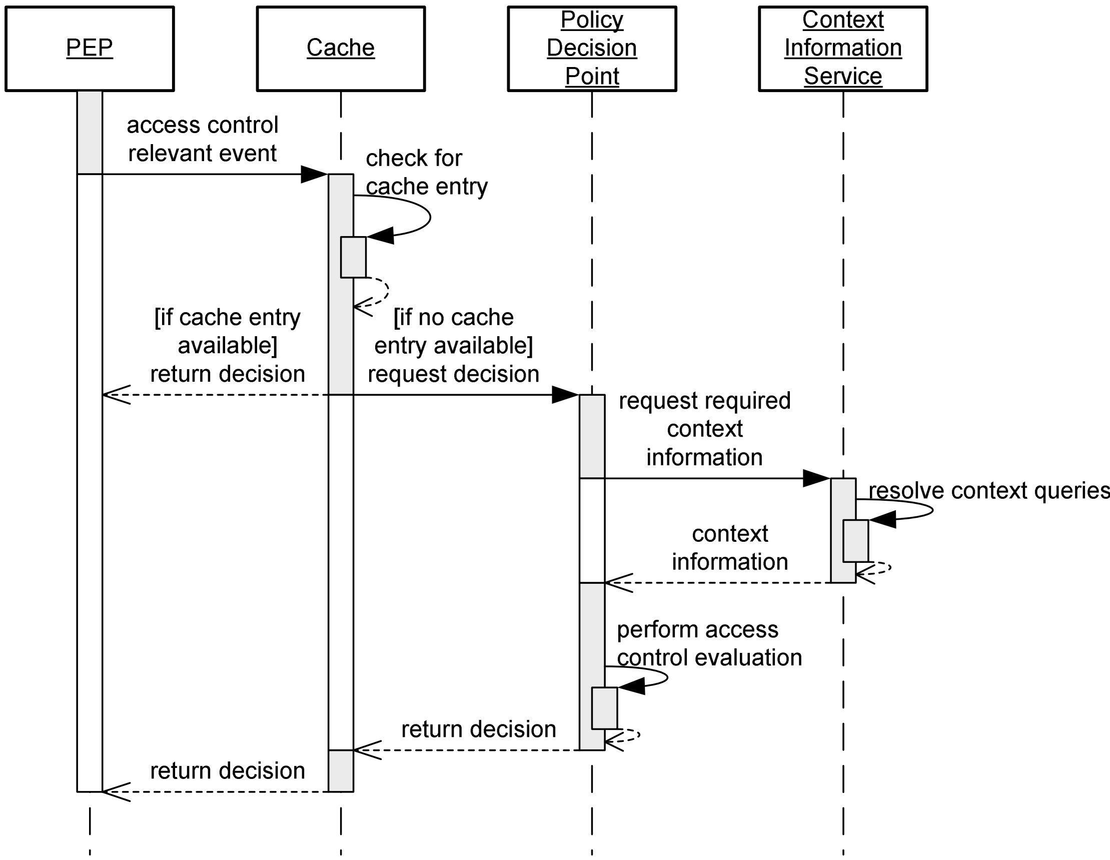


Figure 5-4: Sequence diagram for an access control evaluation using the cache

In summary, our caching strategy requires a cache management component which is responsible for triggering access control request evaluations. The resulting access control decisions are cached until the cache management revokes them.

5.3 Foundations

The complete cache management is build up on a *Caching Heuristic* which defines the relations between system events and respective access control requests for pre-evaluation.

The goal of the heuristic is to define which access control requests have to be pre-evaluated at which events broadcasted during a business process execution, as well as at which events cache entries can be revoked. We use three workflow-specific information sources to gain information at which events to pre-evaluate and revoke cache entries. The information

sources are namely life cycles, business process definitions, and relations between tasks and business objects.

The three sources are introduced in the following section. Afterwards, in Section 5.3.2, we illustrate the details about the structure of the caching heuristic to store this information as rules, such that the cache management may control the evaluation and revocation of access control decisions accordingly.

5.3.1 Workflow-specific Information Sources

Life Cycles

Our caching strategy relies on life cycle events. The goal is to specifically define which of these events are used to trigger pre-evaluations and for which access control relevant events the pre-evaluation should be done.

We already showed that business process-driven systems events correspond to transitions of a life cycle. Hence, it is clear if an instance transits into a state with event *A*, one of the events {*X*, …, *Z*} leaving the state will follow during further execution of the same instance. This means, if it is known that access control decisions for at least one of the events in {*X*, …, *Z*} will be needed, these access control decisions can already be pre-computed at the point in time the preceding event *A* occurred.

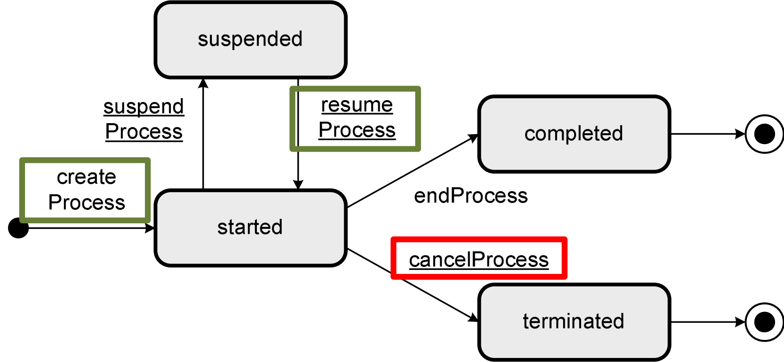


Figure 5-5: The event "cancelProcess" has two preceding events: "createProcess" and "resumeProcess".

In Chapter 3 we have seen that life cycles may contain access control relevant events. These are the events for which access control decisions shall be pre-evaluated. Each of these access control relevant events has one or more preceding events; hence events, which will *always* occur *before* the access control relevant event occurs. Whenever one of these preceding events happens, there is a chance an access control relevant event will follow.

For example, given the process life cycle of JBoss jBPM in Figure 2-6, the event "cancelProcess" is an access control relevant event. Considering the whole life cycle, "cancelProcess" has two preceding events: "createProcess", and "resumeProcess" (illustrated in Figure 5-5; preceding events are coloured in green). After a preceding event occurred, the process instance transits into the state "started", which gives the user the possibility to *cancel* the current process instance. Consequently, we pre-evaluate an access control decision for the access control relevant event "cancelProcess" if either "createProcess" or "resumeProcess" occurs such that an access decision is available if the user decides to abort the process.

Revocations of cached access control decisions should also be triggered by events. There are several events within the life cycles which indicate that a cached access control decision can be removed. If an execution of a process or task is finished, either by a cancellation or a regular termination, its instance transits into a final state. Cached decisions of instances which are finished are obsolete. The occurrence of events leading to final states of the life cycles can be used to trigger the revocation of cache entries.

In the example for the life cycles of JBoss jBPM (cf. Section 2.2.2), events leading to final states are "cancelProcess", "endProcess", "cancelTask", and "endTask".

In essence, the system's life cycles are the first of three possible workflow-specific information sources to specifically determine at which point during a process execution pre-evaluations should happen such that the respective access decisions are available up on the occurrence of an access control relevant event. Life cycles also provide the information at which point during execution cached access decisions become obsolete and can be revoked.

Business Process Definition

The second information source comprises deployed business process definitions. Every process definition specifies a set of tasks and the order in which these tasks are executed. Within the execution path of a process, almost every task has at least one following task; the exceptions, of course, are end tasks.

This means, if one task of a process is executed, it is already known that one or more of the immediate following tasks are executed next. Hence, we pre-compute the respective access control requests required for the upcoming tasks. The set of all immediate upcoming tasks build what we call *Fringe*. Figure 5-6 unterhalb illustrates the fringe for the "Create Change Request" of our running example, the "Project Issue Management" process (cf. Section 2.1.2).

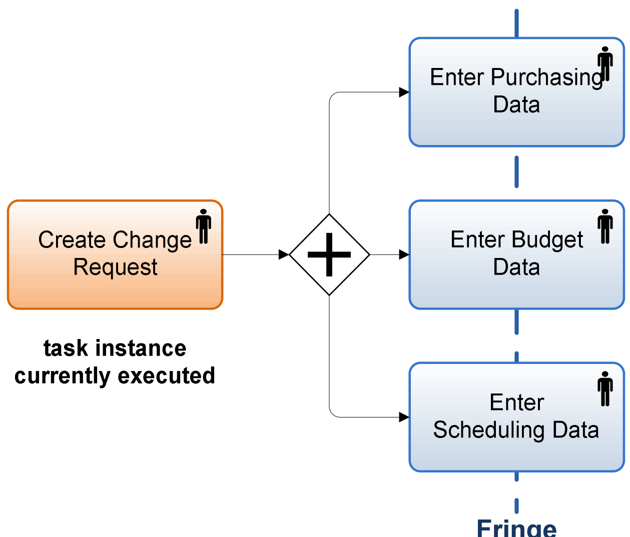


Figure 5-6: Example illustrating immediate upcoming tasks

In general, it would be possible to pre-compute access decisions for all tasks of a business process at once, for instance, at the time a process instance is created. Processes may have a significant amount of tasks and it may have branches leading to several possible paths of execution. Given such a situation, the pre-computation of a complete process at once would, as such, have at least two drawbacks to be taken into account.

Firstly, pre-evaluating all possible access control requests for a complete process might lead to a significant amount of cached access decisions which will never be used. The pre-evaluations as well as the cache entries in the cache, however, require resources (e.g., memory usage, CPU time) which have to be provided by the system.

Secondly, at the point in time a process instance is created, a significant amount of access control requests must be pre-evaluated by the PDP. This will produce bulk-requests and consequently peaks in the system's workload.

In general, however, ProActive Caching allows both: the pre-computation can be limited such that only upcoming tasks are considered (i.e., considering the tasks within the fringe) or can be very comprehensive by pre-evaluating access control requests for the complete process at once.

Business Object Dependencies

The third information source comprises relations between tasks and business objects. The execution of a task usually fetches information from back-end systems (e.g., from an ERP or CRM system) to display it to the user or it requires to create, modify, or delete data in back-end systems (cf. Section 2.2).

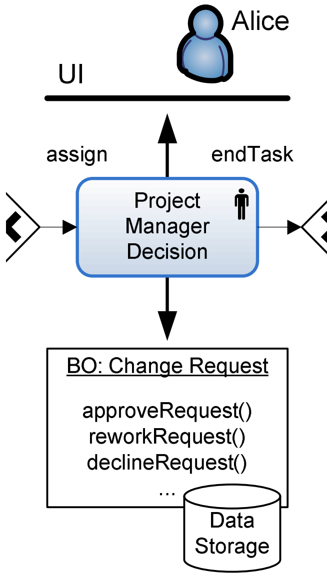


Figure 5-7: Interaction with Business Objects

We described in Section 3.3 that each BO-call requires an access control check against the system's security policy. Although the BO is called by the BPMS during a task instance execution, this is done on behalf of the user who claimed a respective task.

Figure 5-7 illustrates the example that Alice is the manager to decide about a change request. She is assigned to the task "Project Manager Decision", hence, the BO-call for approving, declining, or reworking the request is done on her behalf.

At the moment Alice claims a task instance, it is clear that Alice will be the one for which access control checks for the BO-calls are necessary. Hence, we pre-evaluate the access control requests for the BO-calls as soon as a user claims a task. Furthermore, it is also known that if a task instance is finished with the "endTask" event, cached access control decisions for BO-calls are obsolete and can be revoked from the cache.

Now, as it is clear which system events should trigger access control pre-evaluations, we will present the caching heuristic to specify the relation between events triggering pre-evaluations and the respective access control requests to be evaluated next.

5.3.2 Dependency Relation

The caching heuristic is a set of rules. One rule formalizes the above mentioned dependency between an event triggering a pre-evaluation, and the access request to be pre-evaluated. Such a rule is called *Dependency Relation*.

Figure 5-8: Elements of a Dependency Relation (DR)

A dependency relation contains two elements called *Trigger Target*, and *Successor Request*. Before we define the notion of a dependency relation, we will introduce these two elements in the following sections.

Trigger Target

The *Trigger Target* (TT) defines the point during execution of a process at which a pre-evaluation of an access request should be triggered. A TT specifies a triple, comprising event identifier, resource name, and user name. TTs are compared with events: the target of an event matches with a TT if the respective elements are equal. If this is the case, a pre-evaluation is triggered.

We define the trigger target as follows.

Definition 8 Trigger Target

A *Trigger Target* is a triple TT(*e*, *r*, *u*), where

- *e* is an event identifier,
- *r* is a resource name, and
- *u* is an user name.

Each element is used to match against the corresponding elements of a target of an event. Each of the elements can be specified with an "*"-wildcard, reflecting that any value of the target's corresponding element will lead to a match.

Assume we want to trigger an access control evaluation whenever the user "Alice" starts a new instance of the process "Project Issue Management". An example for a respective TT is

tt_1 = TT(*"createProcess", "Project Issue Management", "Alice"*),

where the elements event identifier, resource name, and user name are specified as values, namely "createProcess", "Project Change Management", and "Alice". A matching event would be

event(*"createProcess", "Project Issue Management", "Alice",* {PIID(*2342*)})

as the event's target elements (*"createProcess", "Project Issue Management",* and *"Alice"*) are equal to the ones specified with tt_1 above.

Alternatively, the elements of a TT can be specified with wildcards. The wildcard "*" helps specifying TTs which do not only match to events with specific event-resource-user-combinations, but also match for events where, for instance, only the event-element is relevant, but the resource and user elements may be arbitrary.

An example for a TT with wildcards is

$$tt_2 = \mathrm{TT}(\text{"createProcess"}, *, *),$$

where only the event is specified, resource and user may contain any value. It states that the tt_2 will match with any event containing the event identifier "createProcess", independent of the resource and user.

Successor Request

The *Successor Request* (SR) defines the access control request for which a pre-evaluation should be done. A dependency relation links the occurrence of an event matching a TT with the pre-evaluation of the SR. Whenever an event matches a TT, the linked SR is sent to the PDP for pre-evaluation. Hence, the evaluation of the access control request *succeeds* the occurrence of a respective matching event.

Definition 9 Successor Request

A *Successor Request* (SR) is a quadruple SR(*e, r, u, piid*), where

- *e* is the identifier of an event,
- *r* is the name of a resource,
- *u* is the name of a user, and
- *piid* is a process instance identifier.

The elements *r, u,* as well as the identifier *piid* may be references, resolving to the actual value(s) (e.g., *r* = *event.*r).

The response of a pre-evaluated SR is the access control decision stored in the cache. As the decision is evaluated by the PDP, the SR must comprise all information required for the PDP to perform an access control request evaluation, namely the event identifier, the resource name, the user name, as well as a respective process instance identifier PIID. We assume that within one process instance, every task instance can be sufficiently identified by the combination of task name and respective process instance identifier. Hence, without loss of generality, we do not include task instance identifiers (TIID) within an SR.

An SR must contain a PIID for which the access control request is pre-evaluated. Access control decisions are pre-evaluated for the same process instance out of which an event triggered the pre-evaluation. Hence, the PIID for the SR is taken from the *event* = event(*n, r, u, I*) which originally triggered the pre-evaluation of the SR.

Recall, we use *event*.X to relate to an event, where X references the respective element of the event. Within an SR, the variable *event* always references to the event triggering the pre-evaluation.

Consider the example that an access decision should be pre-evaluated for the event "cancelProcess", the resource "Project Issue Management", the user "Alice", and for the PIID which is taken from the event triggering the pre-evaluation. The SR would be specified as follows

$$sr_1 = \text{SR}(\textit{"cancelProcess", "Project Issue Management", "Alice", event}.\text{I.piid}).$$

The parameter *event*.I.piid is resolved during runtime, where *event*.I resolves to the set of instance identifiers and *event*.I.piid to the respective process instance id. When the sr_1 is sent to the PDP for evaluation, *event*.I.piid is replaced with the PIID of the triggering event. By using *event*.I.piid, it is guaranteed that the pre-evaluation is performed for exactly that process instance, which actually triggered the evaluation in the first place.

A second example for an SR shows that also other elements than just the PIID can be stated as references.

$$sr_2 = \text{SR}(\textit{"cancelProcess", event}.\text{r}, \textit{event}.\text{u}, \textit{event}.\text{I.piid})$$

sr_2 illustrates an SR for the event "cancelProcess". The resource, the user, as well as the PIID are replaced during runtime with the respective information given from the triggering event. The access request is then sent to the PDP for pre-evaluation.

Dependency Relation

The *Dependency Relation* (DR) links TTs and SRs such that an event matching with a TT starts the pre-evaluation of the specified SR.

We formally define a dependency relation as follows.

Definition 10 Dependency Relation

A *Dependency Relation* (DR) is a tuple DR(*tt, sr*), where

- tt is a Trigger Target, and
- sr is a Successor Request.

If a *target* of an event matches the *tt*, the *sr* is pre-evaluated and the corresponding access decision stored as *Cache Entry*.

The flexibility to determine parts of the SR during runtime enables the possibility to formulate complex and especially more general DRs. For example, assume there are 100 process definitions available in a BPMS. Further assume that a dependency relation should

state that with the creation of a process instance (event: "createProcess"), access control decisions for the event "cancelProcess" should be pre-computed for the same instance.

There are at least two possible ways to accomplish this task. The first one is to define 100 DRs, stating for each process definition that if the event "createProcess" occurs, the SR for "cancelProcess" should be triggered and sent for pre-evaluation. This is illustrated with Table 4.

dr_{P001}	tt_{P001} = TT(*"createProcess", "Process 001", "*"*) sr_{P001} = SR(*"cancelProcess", "Process 001", event.*user, *event.*I.piid)
dr_{P002}	tt_{P002} = TT(*"createProcess", "Process 002", "*"*) sr_{P002} = SR(*"cancelProcess", "Process 002", event.*user, *event.*I.piid)
dr_{P003}	tt_{P003} = TT(*"createProcess", "Process 003", "*"*) sr_{P003} = SR(*"cancelProcess", "Process 003", event.*user, *event.*I.piid)
...	
dr_{P100}	tt_{P100} = TT(*"createProcess", "Process 100", "*"*) sr_{P100} = SR(*"cancelProcess", "Process 100", event.*user, *event.*I.piid)

Table 4: Exemplary DRs for 100 process definitions

The second possibility is to only define one DR. We use the wildcard "*" for the TT such that it gets triggered with the creation of any process given in the system. Further, we use the placeholder "*event*.r" for the SR such that the resource name for the access control request is retrieved during runtime directly from the respective "createProcess"-event. The DR is shown with Table 5.

dr_{Px}	tt_{Px} = TT(*"createProcess", "*", "*"*) sr_{Px} = SR(*"cancelProcess", event.*r, *event.*u, *event.*I.piid)

Table 5: Exemplary generic DR for all process definitions in the system

Such dynamic parameters do not only reduce the number of required dependency relations, but also simplify possible administration overhead. If, for instance, new process definitions are inserted or old process definitions are removed from the system, the DR from Table 5 does not have to be adjusted - in contrary to the ones in Table 4.

In the previous section we illustrated that business process definitions are information sources to determine which access control requests should be pre-evaluated. The idea is to pre-evaluate access control requests for all directly upcoming tasks within the fringe. The goal is to guarantee that with the instance creation of a task of the fringe, the tasks immediate required access control decisions are already available.

For example, directly after the creation of a task instance, the instance is available in the system and dedicated to appear in the GWLs of those users which are allowed to claim it. The GWL determines a user's permission to claim a task by sending an access control request to the PDP, checking whether the user may be assigned to it. Hence, immediately after the creation of a task instance, an access control decision for the access control relevant event "assign" is required, such that we pre-evaluate decisions for the access control relevant event "assign" already upfront at the time an immediate preceding task is created. In other words, with the creation of a task instance, we pre-compute access control decisions for the event "assign" for those tasks within the current task's fringe.

Before we give an example, we have to consider for which users access control requests are pre-evaluated.

The GWL can be accessed by multiple users registered with a BPMS. To guarantee each user a fast access to the GWL, for each user and each upcoming immediate tasks an access decision should be available and, hence, pre-computed. This might become very extensive and possibly requires the computation of many access control decisions for users which actually never or only marginally access a GWL. In consequence, our work provides the possibility to perform access control request pre-evaluations for a pre-defined set of users. The set may contain all users of a system or only those which, for instance, frequently access the GWL. The set of users for which access control decisions should be pre-evaluated is determined, for example, by the PDP, using a function *pdp.*getUsers.

$dr_{Px\text{-}GWL}$	$tt_{Px\text{-}GWL}$ = TT*("createTask", "Create Change Request", "*")* $sr_{Px\text{-}GWL}$ = SR*("assign", "Enter Purchase Data", pdp.*getUsers*, event.*I.piid*)*

Table 6: Exemplary DR illustrating the use of functions to receive a list of users for which access requests are pre-evaluated

Assume the creation of a task instance for "Create Change Request" of our running example "Project Issue Management". With its creation, all access control requests for the immediate succeeding task "Enter Purchase Data" shall be evaluated, for a set of users defined by the function *pdp.*getUsers. The respective DR is given with Table 6.

$dr_{Px\text{-}BO}$	$tt_{Px\text{-}BO}$ = TT*("assign", "Create Change Request", "*")* $sr_{Px\text{-}BO}$ = SR*("createNewRequest", "Change Request", event.*u*, event.*I.piid*)*

Table 7: Exemplary DR for triggering pre-evaluations of access control requests for function calls on business objects

Our caching strategy also allows pre-evaluating access control requests for business object calls. We pre-evaluate access control requests at the moment at which it is clear, which user

will perform the task. This is the case at the point the user claims it and an "assign"-event occurs. An example for a dependency relation is given with Table 7. It states that an access control request is pre-evaluated for the BO-call "createNewRequest" of the BO "Change Request" whenever a user claims the task "Create Change Request".

5.3.3 Cache Entry Revocations

Revoking cached access control decisions is part of the cache management to avoid storing cache entries no longer needed. The information sources we introduced (cf. Section 5.3.1) help not only to define the dependency relations, but also provide the information about the events which state that cache entries are obsolete.

With the termination of a process or task instance, all cached decisions for these instances can be removed from the cache. This means that events which lead to final states of the process and task life cycles may be used to trigger revocations of cache entries for a complete instance.

In this section we will introduce *General Revocation Targets* (GRT) which specify event identifiers which reflect that a process or task instance has terminated. A successful match between a broadcasted event and a GRT revokes all cache entries for the event's respective resource and process instance identifier. A GRT comprises two elements, namely an event identifier which should trigger the revocation, and a resource for which the cache entries should be revoked. Of course, this implies that the resource of the event triggering the revocation is the same resource for which cache entries should be revoked. The GRT's definition and accompanying example is given next.

Definition 11 General Revocation Target

A *General Revocation Target* (GRT) is a tuple GRT(*e*, *r*), where

- *e* is an event identifier, and
- *r* is a resource name.

Both elements are used to match against the corresponding elements of a broadcasted *event* = event(*n, r, u, I*). The resource element can further be specified with an "*"-wildcard reflecting that any value of the event's resource element will lead to a match. A successful match revokes all cache entries for the broadcasted event's respective resource and process instance identifier (i.e., *event*.I.piid).

Assume we want to specify GRTs for the complete "Project Issue Management" process such that whenever a task instance or the complete process instance is terminated, all respective cache entries are removed from the cache. For the JBoss jBPM life cycles, transitions which lead to end states are "cancelProcess", "endProcess", "cancelTask", and "endTask". We use these event identifiers to specify the GRTs. As example, in Table 8 we state GRTs for the process as well as its first and last tasks; GRTs for all other tasks of this process are similar.

grt_1	{GRT(*"cancelProcess", "Project Issue Management"),* GRT(*"endProcess", "Project Issue Management"),* GRT(*"cancelTask", "Create Change Request"),* GRT(*"endTask", "Create Change Request"),* *...* GRT(*"cancelTask", "Project Issue Notification"),* GRT(*"endTask", "Project Issue Notification")}*

Table 8: Examples for a set of GRTs, given the "Project Issue Management" process

Similar to a dependency relation's trigger target, a GRT is matched against the system's life cycle events, broadcasted during process and task execution. A match triggers the revocation. The selection of which entries have to be removed is done by the respective PIID and resource identifier of the triggering event. Given the GRTs from Table 8, the following event would revoke all cache entries for the process instance with the PIID *"2342"* and the resource *"Project Issue Management"*:

event(*"endProcess", "Project Issue Management", "Alice",* {PIID(*2342*)}).

The use of wildcards for the resource identifier - as already introduced for the trigger targets within a dependency relation - reduces administration overhead if new processes are inserted into the system, or already existing ones removed. An example is given with Table 9.

grt_2	{GRT(*"cancelProcess", "*"),* GRT(*"endProcess", "*"),* GRT(*"cancelTask", "*"),* GRT(*"endTask", "*")}*

Table 9: Examples for a set of GRTs, given the "Project Issue Management" process using wildcards. The depicted GRT's match with all process and tasks defined with an BPMS

GRTs may also be combined with dependency relations as we will show in the next section.

5.3.4 Extended Dependency Relation

In this section we introduce *extended Dependency Relations* (extDR). An extDR is a DR which includes *Revocation Targets* (RT). A RT is more specific than a GRT. It is a quadruple specifying an event identifier, a resource name, a user name, and a process instance identifier.

Definition 12 Revocation Target

A Revocation Target is a quadruple RT(*e*, *r*, *u*, *piid*), where

- *e* is an event identifier,
- *r* is an resource identifier,
- *u* is a user name, and
- *piid* is a process instance identifier.

The idea is to link a created cache entry directly with an RT: if an event matches the RT, the linked cache entry is removed.

An extDR defines not only the TT and the SR, but also a set of RTs. Each RT triggers the revocation of exactly those cache entries, that have been pre-computed based on the same extDR. The difference to GRTs, hence, is that RTs within an extDR do not revoke entries for a complete process or task instance, but very selectively revoke exactly that cache entry, which was created based on the SR, specified with the same extDR.

Figure 5-9: Elements of an extended Dependency Relation (extDR)

We give the definition for an extended DR next.

Definition 13 extended Dependency Relation

An *extended Dependency Relation* (extDR) is a tuple extDR(DR(*TT, SR*), *RT*), where DR(*TT, SR*) is a *Dependency Relation* and *RT* is a *Set of Revocation Targets*. Assuming an event e_1 triggering the creation of *Cache Entries* based on the *SR*, an event e_2 matching one of the respective revocation targets within *RT* leads to a revocation of all *Cache Entries* which were created upon event e_1.

As example, the JBoss jBPM life cycle shows that after a process was created (i.e., the event "createProcess" occurs), a user may call "suspendProcess". Assume a cache entry for "suspendProcess" should be created as soon as a process instance is created. Further, the same cache entry should be revoked if either the event "suspendProcess" occurs (implying the process instance is successfully suspended) or the process instance terminates.

Hence, we specify an extDR (see Table 10) which defines that if an event "createProcess" occurs, a cache entry for "suspendProcess" is pre-evaluated. It further defines that if either

of the events "suspendProcess", "cancelProcess", or "endProcess" occurs, the cache entry pre-evaluated for "suspendProcess" is revoked from the cache. Similar to the SR, we use "*event*.X" to reference from an element of an RT to an element *X* of the initial event which triggered the pre-evaluation of the cache entry in the first place.

extDR	*tt* = TT*("createProcess", "*", "*")* *sr* = *SR("suspendProcess", event.*r*, event.*u*, event.*l.piid*)* *rt = {* RT*("suspendProcess", event.*r*, event.*u*, event.*l.piid*),* RT*("cancelProcess", event.*r*, "*", event.*l.piid*),* RT*("endProcess", event.*r*, "*", event.*l.piid*)}*

Table 10: Exemplary extended DR including Revokation Targets (RT)

This allows defining revocations of cache entries at any possible event of the system. The example also illustrates that it is possible to combine a DR and a set of GRTs, and transform them into extDRs.

In essence, revocation of cache entries can either be done using general revocation targets, which allow defining a small group of targets affecting all process and task instances in the system, or it can be done using a much more fine grained definition where a single revocation can be specifically linked to the creation of another specifically pre-computed access control decision. In fact, the given example (Table 10) shows that GRTs can be expressed using their fine grained counterparts RTs, that are part of an extDRs.

5.3.5 Cache Management

In this section we describe the general operations for our cache management. Goal for the cache management is to pre-evaluate access control decisions based on broadcasted events and a set of (extended) dependency relations.

Figure 5-10 depicts the sequence diagram for pre-evaluating access control requests. Starting point for every pre-evaluation is a *trigger event* broadcasted by the BPMS and eventually received by the cache management component. The four steps are described next.

The **first step** for the cache management is to retrieve relevant dependency relations (DR) and general revocation targets (GRT) from the storage where DRs and GRTs are maintained. A DR or GRT is relevant, if the trigger event matches the respective target of a DR's TT or it matches a GRT.

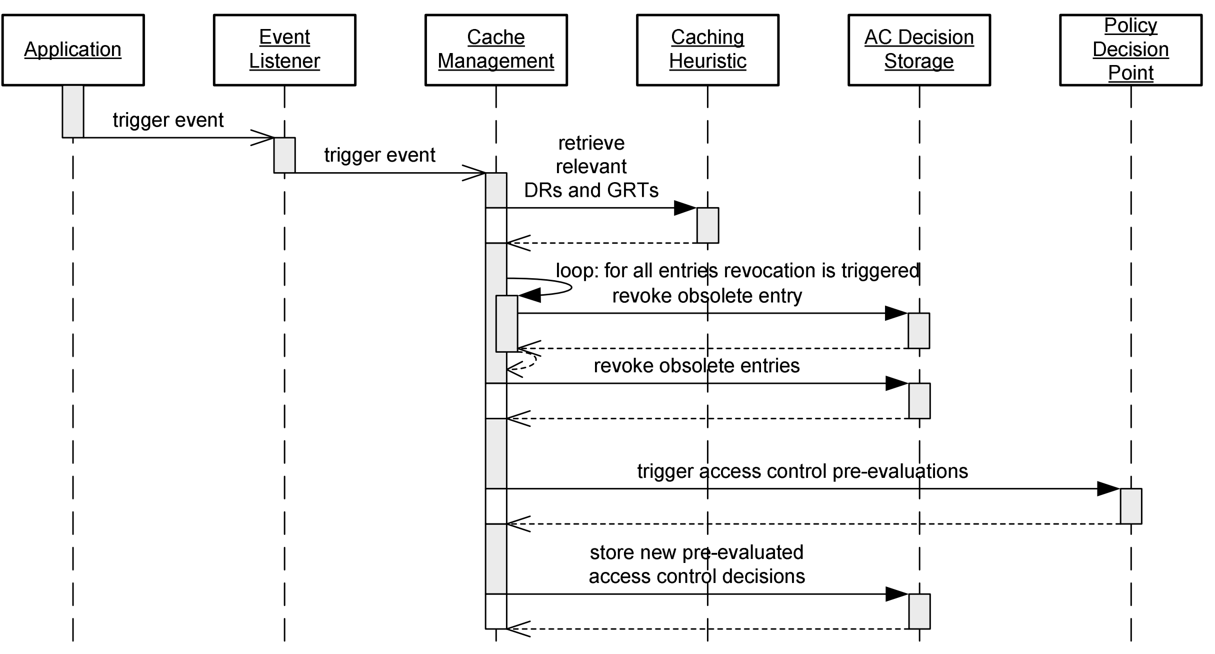


Figure 5-10: Sequence diagram for pre-evaluating and revoking access control decisions

The **second step** revokes obsolete cache entries from the cache. Revocations are triggered if:

1. the trigger event matches a GRT, resulting in a respective revocation of all entries affected by the GRT,
2. the trigger event matches an RT of an extDR, revoking all cache entries which have previously be created, based on the particular extDR's SR, or
3. already stored cache entries are going to be updated by new pre-evaluations.

The **third step** triggers new access control evaluations, using the previously fetched DRs. Hence, the SRs of the DRs are sent to the PDP for evaluation.

Finally, the newly pre-evaluated access control decisions are stored with the cache.

In the next section we will introduce solutions such that access control decisions which are based on dynamic context constraints can be pre-evaluated and cached as well.

5.4 Pre-evaluating, Caching and Updating Dynamic Access Decisions

We previously described that results of access control decisions based on dynamic context constraints may change over time. This happens if the context information changes on which the access control evaluations relied on. Cached access control decisions, hence, may become invalid.

Recall, we generally distinguish between access control constraints which rely on information about a user's past interactions with the system (i.e., change-detectable context constraints) and constraints relying on context information where changes cannot easily be detected (i.e., open constraints).

The difference is that in the first case, we may encounter that the context information changed and the cache management can react accordingly; in the second case, however, context information may change without further notice, such that an instant reaction on a context change is either impossible (e.g., in case of externally located information) or requires significant additional effort for monitoring and tracing data changes in the system.

Nevertheless, for both types of constraints a solution is required, such that our caching strategy can be used in dynamic environments such as business process-driven systems.

In this section we propose solutions for each of the two types, starting with change-detectable context constraints.

5.4.1 Caching Access Decisions based on Change-detectable Context Constraints

In Chapter 3 we introduced "change-detectable context constraints" which comprise constraints types such as dynamic separation of duties (DSoD), binding of duties (BoD), or dynamic cardinality. The goal of this section is to show, how access decisions based on these constraints may be cached.

Recall, user interactions with a BPMS (e.g., claiming a task) are relevant for the evaluation of change-detectable context constraints, as these interactions actually lead to the alternating context information. Each interaction of a user with the BPMS results in an occurrence of events (e.g., "assign"). Hence, if we know which interactions, and consequently which events, lead to context changes, listening to the system's events is enough to recognize that the context has changed.

A context change may render cached decisions invalid, and hence, requires that cached access control decisions are updated. An update involves two steps: (1) invalid access decisions are revoked, and (2) revoked entries are re-evaluated.

Dynamic Separation of Duties

Remember our example process "Travel Request" which we annotated with DSoD constraints (see Figure 3-1) to ensure that no user approves his own travel request, and that two different users approve the request and corresponding budget.

Which user actually performs a task is determined during runtime by a user claiming the task. Given a DSoD constraint as stated above, the user may first be able to choose and claim an arbitrary task out of the set of exclusive tasks, as long as he has the required (role-based)

permissions. As soon as the user claims one of the exclusive tasks, however, he should be restricted to claim further exclusive tasks of the same constraint.

From an access control perspective, the evaluation whether a user is allowed to claim a task is based on the set of tasks the user already claimed (within the same process instance). Hence, it is based on the user's context information.

From a cache management perspective, the interaction of a user claiming an exclusive task may render the cache entries pre-evaluated for the other exclusive tasks invalid, requiring their update. Moreover, if the user revokes from the exclusive task and a new user is assigned to it, the cache entries have to be updated again.

This means, cache updates have to be done if a user's context information changes; the context changes, if a user claims a task or revokes from an exclusive task.

To manage cache updates in general, for one DSoD constraint, we consequently need to know three types of information:

1. The set of system events which may change context information such that updates are necessary,
2. the access control relevant event for which access control decisions should be pre-evaluated or updated, and
3. the set of exclusive tasks within one DSoD constraint.

To 1: At which events the context information relevant for DSoD constraints changes is dependent on the BPMS. For JBoss jBPM the events "assign" and "revoke" reflect that a user-assignment of a task changed.

To 2: In our work, we assume that DSoD constraints must be checked whenever a user claims a task; it must be checked whether he is actually allowed to be assigned to that specific task instance. Hence, "assign" is the respective access control relevant event for which we pre-evaluate access control decisions. In general, the security policy provides that information.

To 3: The set of exclusive tasks is also defined with the DSoD constraint itself. Hence, it can also be found within the security policy.

For clarity, we define a new type of dependency relation (DSoD-DR) specifically for DSoD constraints next.

Definition 14 Dynamic SoD Dependency Relation

A Dynamic SoD Dependency Relation (DSoD-DR) is a tuple DSoD-DR(*contextChangingEvents*, *acrEvent*, *affectedTasks*), where

- *contextChangingEvents* is a set of event identifiers which specify the events at which context information relevant for an DSoD constraint changes (e.g., "assign", reflecting the assignment of a user to a task, and changing the information which tasks the user already executed or executes),
- *acrEvent* is an event identifier which specifies an access control relevant event for which access control decisions have to be pre-evaluated, and
- *affectedTasks* is a set of at least two, but possibly multiple exclusive tasks within a business process definition.

The occurrence of a context changing event *cce* = Event(*e*, *r*, *u*, *piid*) requires that for each task $t \in$ *affectedTasks* an access control request for the user triggering the event *cce* is pre-evaluated (i.e., *sr* = SR(*acrEvent*, *t*, *cce*.u, *cce*.piid)), and the resulting access control decisions are stored in the cache, thereby updating potentially previously stored entries.

The DSoD-DRs define exactly the above mentioned information required for updating cached decisions. The DSoD-DR, however, does not define explicit revocation targets which remove updated cache entries when a task instance is finished. The set of general revocation targets defined with Definition 11 take care of cleaning up after a task instance is finished.

An example of DSoD-DRs for the previously mentioned "Travel Request" process is given with Table 11.

DSoD-DR$_1$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* *affectedTasks = {*Task*("Create Travel Request"),* Task*("Manager Approval")}*
DSoD-DR$_2$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* *affectedTasks = {*Task*("Create Travel Request"),* Task*("Budget Approval")}*
DSoD-DR$_3$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* *affectedTasks = {*Task*("Manager Approval"),* Task*("Budget Approval")}*

Table 11: DSoD-specific Dependency Relations (DSoD-DR) for the "Travel Request" process depicted with Figure 3-1

Recall, DSoD constraints can be generalized by stating that a user may perform *n* tasks out of a set of *m* exclusive tasks (cf. Section 3.1.2). The DSoD constraints of our "Travel Request" example are special cases, where the user may execute 1 task, given a set of 2 exclusive tasks; hence, *n* = 1, and *m* = 2. For the general case, however, the same update strategy holds as for the special case: if the user assigns to one of the exclusive tasks, all cache entries

for the other exclusive tasks have to be updated. In case the user reached the maximum number of allowed tasks, the update of the cache entries of the remaining exclusive tasks will reflect that: the user may not claim any further exclusive tasks. Hence, also for the general case the specific DSoD-DRs contain all necessary information required for pre-evaluating and updating cache entries.

Similar to dependency relations for dynamic separation of duties, in the following section we introduce a new type of dependency relations for binding of duties constraints.

Binding of Duties

In Chapter 3 we introduced the change-detectable context constraint Binding of Duties (BoD). BoD states that a user which performed one task out of a set of tasks, he also must perform the remaining task(s) of the same set.

This means, we must update the cache for all tasks within such a set of bounded tasks as soon as a user claimed one of them. This is similar to DSoD constraints. The difference is that the cache must not only be updated for the one particular user which claimed one of the bounded tasks, but for all users for which access decisions have or will be pre-evaluated, because none of them may claim any of the remaining bounded tasks anymore.

From a cache management perspective - given a user claiming one task out of a set of bounded tasks - we need to update cache entries for all remaining bounded tasks. To achieve this in general, for one BoD constraint, we again need to know three types of information:

1. The set of system events which may change context information such that updates are necessary,
2. the access control relevant event for which access control decisions should be pre-evaluated or updated, and
3. the set of bound tasks within one BoD constraint.

Again, this information is either BPMS-specific (i.e., the set of events changing the context information) or can be found in our fourth information source, the security policy.

We define a new type of dependency relation (BoD-DR) specifically for BoD constraints next.

Definition 15 Binding of Duties Dependency Relation

A Binding of Duties Dependency Relation (BoD-DR) is a triple BoD-DR(*contextChangingEvents*, *acrEvent*, *affectedTasks*), where

- *contextChangingEvents* is a set of event identifiers which specify the events at which context information relevant for an BoD constraint changes (e.g., "assign", reflecting the assignment of a user to a task, and changing the information which tasks the user already executed or executes),
- *acrEvent* is an event identifier which specifies an access control relevant event for which access control decisions have to be pre-evaluated, and
- *affectedTasks* is a set of at least two, but possibly multiple, mutually bounded tasks within a business process.

The occurrence of a context changing event *cce* = Event(*e*, *r*, *u*, *piid*) requires that for each task $t \in$ *affectedTasks* access control requests for all affected users (i.e., *sr* = SR(*acrEvent*, *t*, *pdp*.getUsers, *cce*.piid)) are pre-evaluated and the resulting access control decisions are stored in the cache, thereby updating potentially previously stored entries.

The dependency relation defines exactly the information required to perform the necessary updates of cache entries. Recall our running example "Project Issue Management" with the three bounded tasks "Create Change Request", "Project Manager Decision", and "Project Issue Notification" depicted in Figure 3-2. All three tasks should be processed by the same user such that all of them belong to one set of bounded tasks. This is illustrated with an exemplary BoD-DR next.

BoD-DR$_1$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* affectedTasks = *{*Task*("Create Change Request"),* Task*("Project Manager Decision"),* Task*("Project Issue Notification")}*

Table 12: BoD specific Dependency Relation for the BoD example depicted with Figure 3-2

Similar to the DSoD-DRs, the events "assign" and "revoke" stated in the respective element *contextChangingEvents* are based on the events of the workflow engine of JBoss jBPM. The same assumption holds that with the interaction of claiming or revoking a task, a respective BoD constraint is affected and cache entries have to be updated.

Dynamic Cardinality

For the change-detectable context constraint *Dynamic Cardinality* holds the same as for DSoD and BoD: a user becomes restricted to perform a task based on the information how often a user already claimed and executed a certain task within the same process instance.

Hence, the permission for a user to claim another instance of a task may change whenever the user claims a task instance or revokes from an instance.

We define a specific type of dependency relation (DCard-DR) for Dynamic Cardinality constraints.

Definition 16 Dynamic Cardinality Dependency Relation

A Dynamic Cardinality Dependency Relation (DCard-DR) is a triple DCard-DR(*contextChangingEvents* , *acrEvent*, *affectedTask*), where

- *contextChangingEvents* is a set of event identifiers which specify the events at which context information relevant for an dynamic cardinality constraint changes (e.g., "assign", reflecting the assignment of a user to a task, and changing the information which task instance the user already executed or executes),
- *acrEvent* is an event identifier which specifies an access control relevant event for which access control decisions have to be pre-evaluated, and
- *affectedTask* is one task within a business process affected by a dynamic cardinality constraint.

The occurrence of a context changing event *cce* = Event(*e*, *r*, *u*, *piid*) requires that for the affected task *t* access control requests for the user triggering the event *cce* are pre-evaluated (i.e., *sr* = SR(*acrEvent*, *t*, *cce*.u, *cce*.piid), and that the resulting access control decision is stored in the cache, thereby updating a potentially previously stored entry.

As an example, we assume that for our process "Project Issue Management" there are a limited number of iterations a project manager is allowed to rework the change request, until the request is cancelled. This requires a dynamic cardinality constraint on the task "Project Manager Decision" for which we present a corresponding DCard-DR with Table 13.

DCard-DR$_1$	*contextChangingEvents = {"assign", "revoke"}* acrEvent = *{"assign"}* *affectedTask* = *{*Task*("Project Manager Decision")}*

Table 13: Dynamic Cardinality-specific Dependency Relation for the process "Project Issue Management" depicted with Figure 2-2

5.4.2 Caching Access Control Decisions based on Attribute-based Constraints

There are types of constraints which require dynamic context information for which context changes cannot be detected by listening to events. In Chapter 3 we generally classified them as *open constraints* (OC). This includes attribute-based constraints which are dependent on, for example, environmental or system attributes (e.g., the current time, return values of external services), or subject attributes (e.g., the location of a user).

It also holds for these constraints that access control decision may become invalid over time if the context changes. Hence, caching of access decisions based on such context information is not feasible without further measures.

Recall, we assume that all access control decisions are based on the evaluation of (role-based) permissions defined with the security policy. Permissions, however, may be restricted by multiple constraints, including open constraints. Permissions as well as all constraints have to be evaluated to come to a final decision. If all constraints hold, the permission's effect (i.e., PERMIT) is returned. If one or more constraints are not satisfied, they restrict the permission and DENY is returned.

OCs are usually expressed by a set of conditions which must be satisfied by evaluating to *true* (cf. Section 3.1.5). For example, the constraint stating that the current time must be within working hours (i.e., "6 a.m. ≤ Now( )" and "Now( ) ≤ 8 p.m.") have to evaluate to *true*. If the conditions evaluate to true, the overall constraint is satisfied; if the conditions evaluate to false, the constraint restricts the permission.

For our caching strategy, a cache entry is a stored response from a request which was sent to the PDP by the cache management component: it contains the final decision of a PDP's request evaluation at the point in time the cache management component triggered the evaluation. This includes equally the evaluation of permissions, as well as the evaluation of all dynamic context constraints, required to come to a final access control decisions. In contrary to change-detectable context constraints, the result of an OC can only be considered valid at the point in time of its evaluation.

In consequence, for OCs, we do not use the actual available context information, but we assume that all conditions of OCs evaluate to true (no matter of the values the actual dynamic context information during evaluation holds). Hence, given the above example, the access control decision *ce*.d stored in a cache entry *ce* assumes that the current time lies between 6 a.m. and 8 p.m., independent of the actual time at the point the access control evaluation was triggered and performed.

If the cache entry is eventually used to answer an access control request (see Figure 5-11 for illustration), then the OC is checked against the real values of the current context information. Hence, given the above example, at the point the cache entry is consumed, the current time is used to check whether the OC is satisfied, hence, whether the current time is within working hours. If this is the case, and all other OCs also evaluate to true, the stored access control decision *ce.*d holds (as during the decision's evaluation the OCs were equally assumed to evaluate to true) and *ce.*d is returned.

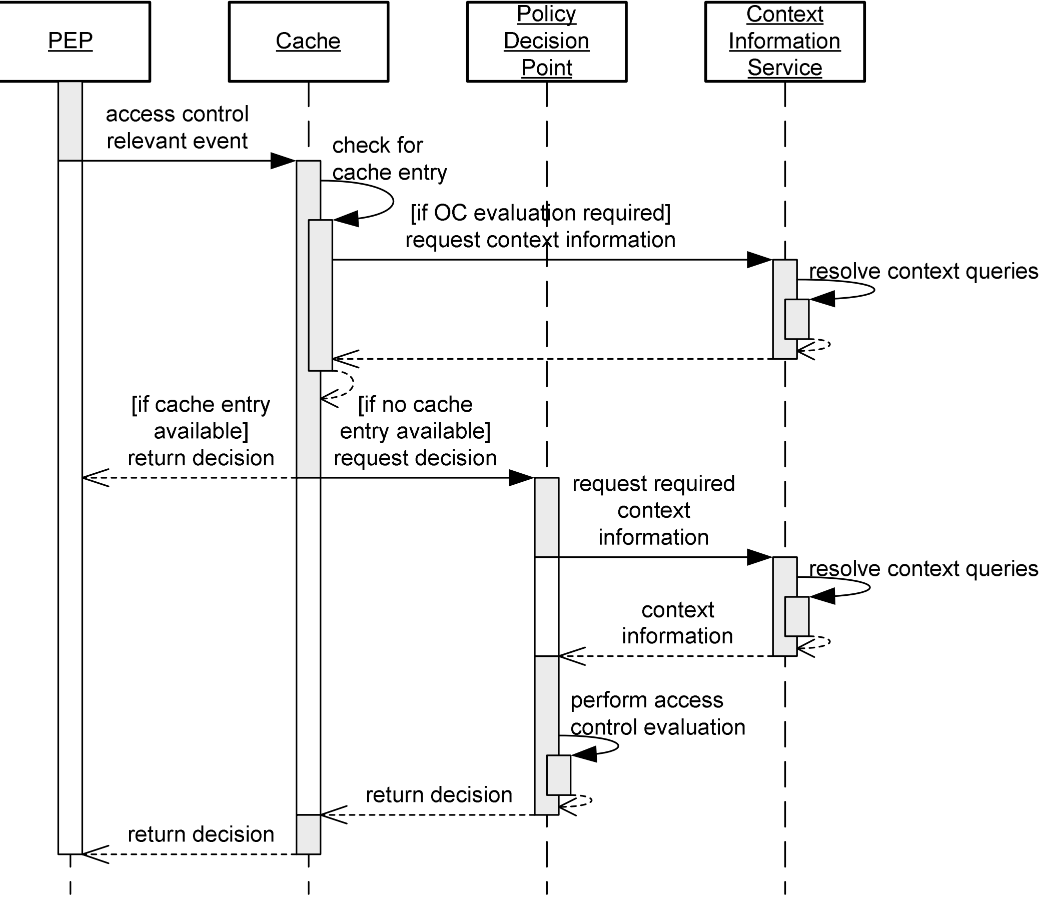


Figure 5-11: Sequence diagram for access control request processing using open constraints

If one of the OCs evaluates to false, it depends on the security policy and its evaluation algorithm, whether just the opposite result (i.e., usually DENY) can be returned, or a complete regular evaluation with the PDP must be done. If the security policy, for instance, states that all single access evaluations must result in granting access such that the overall result is PERMIT (as we assumed above), the failure of a single OC not evaluating to true would automatically result in an overall DENY which the cache can instantly provide without further, regular evaluation.

For the evaluation of OCs within the cache it is necessary that the cache entry contains an additional attribute storing the OC. We define *OC-aware Cache Entries* as follows.

Definition 17 Open Constraint-aware Cache Entry

A *Open Constraint-aware Cache Entry* is a tuple OCCE(CE(*e, r, u, piid, d), oc*), where CE(*e, r, u, piid, d*) is a *Cache Entry* and *oc* is a set of conditions, called *Open Constraints*. Each of these conditions has to evaluate to *true* such that the access control decision *d* holds as overall result.

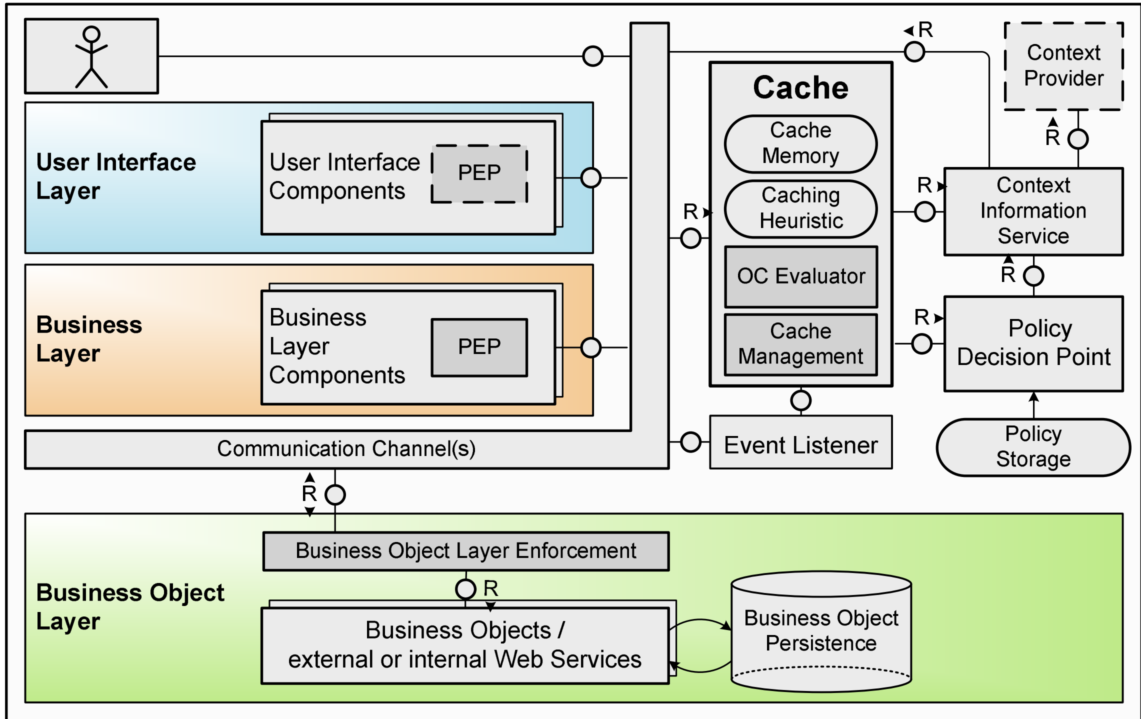


Figure 5-12: Caching Architecture with an extended Cache including an Open Constraint Evaluation component.

Furthermore, the cache component previously illustrated in Figure 5-3 must be extended as shown in Figure 5-12. The cache component requires an additional *open constraint evaluation component* (short OC Evaluator) which is able to evaluate the conditions stored with an OC-aware cache entry. The evaluation of open constraints further requires access to the dynamic context information based on which the OC should be evaluated. Hence, we connect the cache to the same context information service the PDP already uses for its regular access control evaluations.

5.5 Summary

In this chapter we introduced our ProActive Caching strategy. The strategy's goal is to anticipate and pre-evaluate access control requests required during the execution of a business process instance such that they are already available in the cache when eventually needed.

ProActive Caching is based on a caching heuristic consisting of dependency relations. Dependency relations define events which trigger access control request evaluations. The results are stored in a cache. Using the caching heuristic, it can exactly be defined at which point during a process execution, pre-evaluations should be triggered. Additionally, it is possible to define at which point previously computed cache entries should be revoked. We

showed four information sources, namely BPMS life cycles, process definitions, relations between tasks and business objects, as well as the security policy, which help to define dependency relations.

Furthermore, not all access control decisions can easily be cached as they are based on dynamic context information. Dynamic context information may change over time, possibly rendering cached decisions invalid. There are two types of constraints. One of them relies on dynamic context information which only changes based on user interactions with the BPMS. We call them change-detectable context constraints. The other type relies on context information where changes can only be detected spending significant overhead or it is not possible at all, namely open constraints. We described solutions for both types of constraints such that caching is still feasible.

In the following chapter we illustrate how the caching heuristics can be generated automatically, using the mentioned information sources.

6 Caching Heuristic

The pre-evaluation of access control requests is based on a caching heuristic comprising a set of dependency relations (DR). It is cumbersome and error-prone to generate a caching heuristic manually, especially if the workflow system implements a large amount of business processes.

In this chapter we will illustrate in detail how the caching heuristic for our caching strategy can be generated automatically, derived from the four information sources

- life cycles of a business process management system (BPMS),
- business process definitions,
- relations between tasks and business objects, and
- the security policy, including its dynamic context constraints.

We will first present the respective algorithms for computing the different DRs based on the life cycles and the process definition. Afterwards, we present an algorithm for generating DRs based on the system's security policy as well as an algorithm for generating respective revocation targets.

6.1 Preliminaries

In our presentation, we roughly follow the pseudocode conventions of [13]:

- The symbol _ ← _ denotes variable assignments.
- Variables are local to the given function.
- Compound data, for instance, life cycles, are organised in objects or structures, which comprise attributes or fields. For example, we write *out*[*n*] (*in*[*n*]) for accessing the set of outgoing (incoming) events of state *n* and *src*[*e*] (*dest*[*e*]) for accessing the source (destination) state of event *e*.
- We use the notations from Definition 2, for instance, Q(*TLC*) refers to the states of the task life cycle *TLC*; F(*TLC*) refers to the end states of a task life cycle.
- Parameters are passed to a function by value.
- Fixed values are expressed in single quotes (e.g., '*pdp*.getUsers', '*event*.I.piid', or '*').

We assume that life cycles and process definitions are stored in data structures which allows for efficient access to the states (or 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 events of a given state directly, and that we can access the source state of an event.

Recall, that there are events which require access control checks (i.e., access control relevant events). This especially includes events of life cycles which we formally defined with Definition 2. For our algorithms we use the previous definition and define an access control aware formal life cycle model which explicitly expresses a set of all access control relevant events within the life cycle.

Definition 18 Access Control Aware Formal Life Cycle Model

A *access control aware formal life cycle model* is a sextuple $(Q, \Sigma, \Sigma_{AC}, q_0, \delta, F)$, where $(Q, \Sigma, q_0, \delta, F)$ is a life cycle and $\Sigma_{AC} \subseteq \Sigma$ is a finite set of access control relevant events to be considered for access control pre-evaluations.

We assume that it is known, which life cycle events are access control relevant prior to the automatic generation of our caching heuristic such that Σ_{AC} is available. The set Σ_{AC} can, for instance, be determined automatically in a pre-computation step by analysing the security policy and mark every event occurring in the policy as access control relevant.

For our exemplary process life cycle (see Figure 2-7) this results in a formal access control aware life cycle model $(Q, \Sigma, \Sigma_{AC}, q_0, \delta, F)$, where

- $Q = \{s_{inactive}, s_{init}, s_{suspended}, s_{fail}, s_{end}\}$,
- Σ = {createProcess, suspendProcess, resumeProcess, cancelProcess, endProcess},
- Σ_{AC} = {suspendProcess, resumeProcess, cancelProcess},
- $q_0 = s_{inactive}$,
- δ = {($s_{inactive}$, createProcess) $\mapsto$ s_{init}, (s_{init}, suspendProcess) $\mapsto$ $s_{suspended}$, ($s_{suspended}$, resumeProcess) $\mapsto$ s_{init}, (s_{init}, cancelProcess) $\mapsto$ s_{fail}, (s_{init}, endProcess) $\mapsto$ s_{end}}, and
- $F = \{s_{fail}, s_{end}\}$.

Note, the event "createProcess" is not part of the set of Σ_{AC}. We do not plan to cache access control decisions for this event. In our model, the event "createProcess" is independent of any previous events such that there is no event which could trigger pre-evaluations for the creation of a process instance. If important for the system's performance, however, the initialization of the cache, as well as the deployment and revocation of (new) process definitions could be used to trigger respective pre-evaluations.

For our exemplary task life cycle (see Figure 2-9) the formal access control aware life cycle model is ($Q, \Sigma, \Sigma_{AC}, q_0, \delta, F$), where

- Q = {$s_{inactive}$, s_{init}, s_{start}, $s_{suspended}$, s_{fail}, s_{end}},
- Σ = {createTask, assign, revoke, startTask, suspendTask, resumeTask, cancelTask, endTask},
- Σ_{AC} = {assign, cancelTask},
- q_0 = $s_{inactive}$,
- δ = {($s_{inactive}$, createTask) $\mapsto$ s_{init}, (s_{init}, cancelTask) $\mapsto$ s_{end}, (s_{init}, revoke) $\mapsto$ s_{init}, (s_{start}, revoke) $\mapsto$ s_{start}, (s_{init}, assign) $\mapsto$ s_{init}, (s_{start}, assign) $\mapsto$ s_{start}, (s_{init}, startTask) $\mapsto$ s_{start}, (s_{start}, suspendTask) $\mapsto$ $s_{suspended}$, ($s_{suspended}$, resumeTask) $\mapsto$ s_{start}, (s_{start}, cancelTask) $\mapsto$ s_{fail}, (s_{start}, endTask) $\mapsto$ s_{end}}, and
- F = {s_{fail}, s_{end}}.

Note that also the event "revoke" is not considered as access control relevant event. We explicitly model the event in our formal task life cycle (cf. Figure 2-9) as it is broadcasted by the BPMS; "revoke" is, however, no event which could explicitly be called by a user - it is only executed in conjunction with the assignment of a new user to a task. The assignment, however, is already checked against the security policy. Hence, no access control check is performed for the revocation event of a user such that Σ_{AC} does not list it.

6.2 Static Access Control

For our DR algorithms, we will use the introduced process and task life cycles (see Figure 2-7 and Figure 2-9), as well as the process definition for the "Travel Request" (see Figure 1-1) as examples.

Recall, a DR is a tuple DR(*tt, sr*) where *tt* is a trigger target (TT) and *sr* is a successor request (SR):

dr	*tt* = TT(*event, resource, user*) *sr* = SR(*access control relevant event, resource, user, piid*)

This corresponds to a *mapping* of a TT to an SR:

TT(*event, resource, user*) $\mapsto$ SR(*access control relevant event, resource, user, piid*),

where the event specified with the TT must precede the access control relevant event of the SR. Both events are bound to a respective resource, i.e., a process or a task as well as respective user and PIID. In this section, we will use the just stated mapping notation for DRs as this notation is more convenient to be expressed within the algorithms presented next.

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.
- *GenDR−PD* (Listing 2) is the second algorithm. 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.
- *GenDR* (Listing 3) is the third algorithm. This algorithm orchestrates the overall generation of the DRs. 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.

We will go through each of the algorithms next.

GenDR−LC

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 (cf. 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 our initializing event (*iEvent, iRes, '*'*) or the incoming events (*i*, *res, '*'*) to the outgoing access control relevant events (*o*, *res, user, piid*) (Listing 1, lines 3–14) and store them as either

DR_1: TT(*iEvent, iRes, '*'*) ↦ SR(*o*, *res,* '*pdp*.getUsers', '*event*.I.piid') (see Listing 1, line 7), or

DR_2: TT(*i*, *res, '*'*) ↦ SR(*o*, *res, 'event*.user', '*event*.I.piid') (see Listing 1, line 9).

Note that we use wildcards '*' for the element *user* with incoming events such that TTs remain independent of a specific user; and we use '*pdp*.getUsers' or *'event*.user' for the element *user* with outgoing events (i.e., for SRs), depending on whether it is required that cache entries must be created for multiple users or just one user.

```
 1:  function GenDR−LC(LC, res, (iEvent, iRes, '*'))
 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 ← TT(iEvent, iRes, '*') ↦ SR(o, res, 'pdp.getUsers', 'event.I.piid')
 8:                  else
 9:                      entry ← TT(i, res, '*') ↦ SR(o, res, 'event.user', 'event.I.piid')
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

DR_1 creates SRs which pre-evaluate cache entries for a list of users. Recall, cache entries for multiple users are always necessary if the access control relevant event may be called by a list of users. The general worklist (GWL), for instance, checks for each user entering its GWL, whether a task available in the system may be claimed by that particular user[4].

For JBoss jBPM holds that all access control relevant events which *leave the first state* of a life cycle (see if-statements Listing 1, line 5 and 6) are events which are possibly called by multiple users, such that we create SRs as shown with DR_1 (cf. Listing 1, line 7); all other access control relevant events must only be created for the user actually assigned to a task instance which is reflected by DR_2 (cf. Listing 1, line 9).

It also holds that the event which triggers the pre-computation for those access control relevant events which *leave the first state* of a life cycle lies in the event that happens *before* a task instance is created. There are two scenarios for such events.

The **first scenario** is that the task for which DRs should be created 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 instance itself. Hence, the trigger for creating access control decisions for the first task of a process is the "createProcess" event for the process instance.

[4] see Section 2.2.1 for a detailed description of the GWL

We use the above mentioned third argument of the algorithm GENDR−LC and initialize the algorithm with the "createProcess"-event of the process (*$createProcess_{PLC}$, res, '*'*).

For example, to generate all DRs for the first task "Create Travel Request" in our "Travel Request" process (cf. Figure 1-1), we call the algorithm as follows:

GENDR-LC(*TLC*, *"Create Travel Request"*, (*"$createProcess_{PLC}$", "Travel Request", '*'*)).

This results in the set of DRs listed in Table 14, where our initializing event (*"$createProcess_{PLC}$", "Travel Request", '*'*) maps to the two access control relevant events *leaving the first state* of jBPM's task life cycle; namely "assign" and "cancelTask". The other DRs are generated by further iterating and processing through all other states of the life cycle.

$DrSet_{CreateTravelRequest}$	{ *TT("$createProcess_{PLC}$", "Travel Request", "*") ↦* *SR("$assign_{TLC}$", "Create Travel Request", pdp.*getUsers*, event.*I.piid*),* *TT("$createProcess_{PLC}$", "Travel Request", "*") ↦* *SR("$cancelTask_{TLC}$", "Create Travel Request", pdp.*getUsers*, event.*I.piid*),* *TT("$revoke_{TLC}$", "Create Travel Request", "*") ↦* *SR("$assign_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$revoke_{TLC}$", "Create Travel Request", "*") ↦* *SR("$cancelTask_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$assign_{TLC}$", "Create Travel Request", "*") ↦* *SR("$assign_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$assign_{TLC}$", "Create Travel Request", "*") ↦* *SR("$cancelTask_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$startTask_{TLC}$", "Create Travel Request", "*") ↦* *SR("$assign_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$startTask_{TLC}$", "Create Travel Request", "*") ↦* *SR("$cancelTask_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$resumeTask_{TLC}$", "Create Travel Request", "*") ↦* *SR("$assign_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*),* *TT("$resumeTask_{TLC}$", "Create Travel Request", "*") ↦* *SR("$cancelTask_{TLC}$", "Create Travel Request", event.*user*, event.*I.piid*)}*

Table 14: Generated DRs for the first task "Create Travel Request" of our Travel Request process given in Chapter 1

The **second scenario** is that a task for which DRs should be created is within a process definition. In this case, the event which precedes the creation of the task instance is the instance creation of the task that precedes the current one. Hence, the trigger for creating access control decisions for a task within in a process definition is the "createTask"-event of its preceding task. We again use the above mentioned third argument of the algorithm GENDR−LC for initializing the algorithm with the "createTask"-event of the preceding task.

For example, assume the task "Manager Approval". Its preceding task is "Create Travel Request". Hence, the initializing event is (*"createTask$_{TLC}$", "Create Travel Request", '*'*), and we call the algorithm to generate all DRs for the task "Manager Approval" as follows:

GENDR-LC(*TLC*, *"Manager Approval"*, (*"createTask$_{TLC}$", "Create Travel Request", '*'*)).

This results in the set of DRs given with Table 15.

DrSet$_{ManagerApproval}$	*{TT("createTask$_{TLC}$", "Create Travel Request", "*") ↦* *SR("assign$_{TLC}$", "Manager Approval", pdp.*getUsers*, event.*I.piid*),* *TT("createTask$_{TLC}$", "Create Travel Request", "*") ↦* *SR("cancelTask$_{TLC}$", "Manager Approval", pdp.*getUsers*, event.*I.piid*),* *TT("revoke$_{TLC}$", "Manager Approval", "*") ↦* *SR("cancelTask$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("revoke$_{TLC}$", "Manager Approval", "*") ↦* *SR("assign$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("assign$_{TLC}$", "Manager Approval", "*") ↦* *SR("assign$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("assign$_{TLC}$", "Manager Approval", "*") ↦* *SR("cancelTask$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("startTask$_{TLC}$", "Manager Approval", "*") ↦* *SR("assign$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("startTask$_{TLC}$", "Manager Approval", "*") ↦* *SR("cancelTask$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("resumeTask$_{TLC}$", "Manager Approval", "*") ↦* *SR("assign$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*),* *TT("resumeTask$_{TLC}$", "Manager Approval", "*") ↦* *SR("cancelTask$_{TLC}$", "Manager Approval", event.*user*, event.*I.piid*)}*

Table 15: Generated DRs for the task "Manager Approval" of our Travel Request process given in Chapter 1

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., for our "Travel Request" process the call for the algorithm looks as follows:

GENDR−LC(*PLC, "Travel Request"*, (*"createProcess$_{PLC}$", "Travel Request", '*'*)).

Given the process life cycle of JBoss jBPM, calling the algorithm results in the set of DRs listed with Table 16.

$DrSet_{TravelRequest-PLC}$	{TT("createProcess$_{PLC}$", "Travel Request", "*") ↦ SR("cancelProcess$_{PLC}$", "Travel Request", pdp.getUsers, event.I.piid), TT("createProcess$_{PLC}$", "Travel Request", "*") ↦ SR("suspendProcess$_{PLC}$", "Travel Request", pdp.getUsers, event.I.piid), TT("resumeProcess$_{PLC}$", "Travel Request", "*") ↦ SR("cancelProcess$_{PLC}$", "Travel Request", pdp.getUsers, event.I.piid), TT("suspendProcess$_{PLC}$", "Travel Request", "*") ↦ SR("resumeProcess$_{PLC}$", "Travel Request", event.user, event.I.piid)}

Table 16: Generated DRs using the process life cycle as basis, given our Travel Request process in Chapter 1

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):

$$|DrSet_{LC}| \leq \sum_{s \in Q(LC)} |in[s]| \cdot |out[s] \cap \Sigma_{AC}(LC)|$$

GenDR−PD

The second algorithm given with Listing 2 generates all DRs for the tasks of one process. 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 "createTask" event of its preceding tasks.

```
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:                  src ← name[PD]
8:              else
9:                  events ← out[q0(src[i])]
10:                 src ← name[src[i]]
11:             end if
12:             for all e ∈ events do
13:                 DrSet_PD ← DrSet_PD ∪ GENDR−LC(TLC, name[t], (e, src,'*'))
14:             end for
15:         end for
16:     end for
17:     return DrSet_PD
18: end function
```

Listing 2: DR Generation: Process Definition

Calling the function GENDR−PD for our running example, i.e., GENDR−PD*(PD*, *PLC*, *TLC)* results in the DRs listed with Table 17, where *PD* stands for the process definition of the "Travel Request" process.

Note that the last task "Summary/Notification" of the "Travel Request" process is an automated task. We assume that automated tasks do not require pre-evaluated access control decisions such that we skip automated tasks.

$DrSet_{TravelRequest\text{-}PD}$	{TT("createProcess$_{PLC}$", "Travel Request", "*") ↦ SR("assign$_{TLC}$", "Create Travel Request", *pdp*.getUsers, *event*.I.piid), TT("createProcess$_{PLC}$", "Travel Request", "*") ↦ SR("cancelTask$_{TLC}$", "Create Travel Request", *pdp*.getUsers, *event*.I.piid), TT("revoke$_{TLC}$", "Create Travel Request", "*") ↦ SR("assign$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("revoke$_{TLC}$", "Create Travel Request", "*") ↦ SR("cancelTask$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("assign$_{TLC}$", "Create Travel Request", "*") ↦ SR("assign$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("assign$_{TLC}$", "Create Travel Request", "*") ↦ SR("cancelTask$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("startTask$_{TLC}$", "Create Travel Request", "*") ↦ SR("assign$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("startTask$_{TLC}$", "Create Travel Request", "*") ↦ SR("cancelTask$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("resumeTask$_{TLC}$", "Create Travel Request", "*") ↦ SR("assign$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("resumeTask$_{TLC}$", "Create Travel Request", "*") ↦ SR("cancelTask$_{TLC}$", "Create Travel Request", *event*.user, *event*.I.piid), TT("createTask$_{TLC}$", "Create Travel Request", "*") ↦ SR("assign$_{TLC}$", "Manager Approval", *pdp*.getUsers, *event*.I.piid), TT("createTask$_{TLC}$", "Create Travel Request", "*") ↦ SR("cancelTask$_{TLC}$", "Manager Approval", *pdp*.getUsers, *event*.I.piid), TT("revoke$_{TLC}$", "Manager Approval", "*") ↦ SR("cancelTask$_{TLC}$", "Manager Approval", *event*.user, *event*.I.piid), TT("revoke$_{TLC}$", "Manager Approval", "*") ↦ SR("assign$_{TLC}$", "Manager Approval", *event*.user, *event*.I.piid), TT("assign$_{TLC}$", "Manager Approval", "*") ↦ SR("assign$_{TLC}$", "Manager Approval", *event*.user, *event*.I.piid), TT("assign$_{TLC}$", "Manager Approval", "*") ↦ SR("cancelTask$_{TLC}$", "Manager Approval", *event*.user, *event*.I.piid), TT("startTask$_{TLC}$", "Manager Approval", "*") ↦ SR("assign$_{TLC}$", "Manager Approval", *event*.user, *event*.I.piid), TT("startTask$_{TLC}$", "Manager Approval", "*") ↦ SR("cancelTask$_{TLC}$", "Manager Approval", *event*.user, *event*.I.piid),

	TT("resumeTask$_{TLC}$", "Manager Approval", "")* ↦ *SR("assign$_{TLC}$", "Manager Approval", event*.user*, event*.I.piid*),* *TT("resumeTask$_{TLC}$", "Manager Approval", "*")* ↦ *SR("cancelTask$_{TLC}$", "Manager Approval", event*.user*, event*.I.piid*),* *TT("createTask$_{TLC}$", "Create Travel Request", "*")* ↦ *SR("assign$_{TLC}$", "Budget Approval", pdp*.getUsers*, event*.I.piid*),* *TT("createTask$_{TLC}$", "Create Travel Request", "*")* ↦ *SR("cancelTask$_{TLC}$", "Budget Approval", pdp*.getUsers*, event*.I.piid*),* *TT("revoke$_{TLC}$", "Budget Approval", "*")* ↦ *SR("cancelTask$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("revoke$_{TLC}$", "Budget Approval", "*")* ↦ *SR("assign$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("assign$_{TLC}$", "Budget Approval", "*")* ↦ *SR("assign$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("assign$_{TLC}$", "Budget Approval", "*")* ↦ *SR("cancelTask$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("startTask$_{TLC}$", "Budget Approval", "*")* ↦ *SR("assign$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("startTask$_{TLC}$", "Budget Approval", "*")* ↦ *SR("cancelTask$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("resumeTask$_{TLC}$", "Budget Approval", "*")* ↦ *SR("assign$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*),* *TT("resumeTask$_{TLC}$", "Budget Approval", "*")* ↦ *SR("cancelTask$_{TLC}$", "Budget Approval", event*.user*, event*.I.piid*)}*

Table 17: Generated DRs by calling GENDR−PD*(PD, PLC, TLC)* with our Travel Request process given in Chapter 1 and the life cycles of JBoss jBPM

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. 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:

$$|DrSet_{PD}| \leq \left(\sum_{t \in Q(PD)} |in[t]| \right) \cdot |DrSet_{LC}|$$

GenDR

Finally, we generate the complete set of DRs for one process by calling 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. For our "Travel Request" process, the result is a joint set of $\text{DrSet}_{\text{TravelRequest-PLC}}$ and $\text{DrSet}_{\text{TravelRequest-PD}}$.

```
1:  function GENDR(PD, PLC, TLC)
2:      DrSet_CT ← ∅
3:      for all e ∈ out[q_0(PLC)] do
4:          DrSet_CT ← DrSet_CT ∪ GENDR-LC(PLC, name[PD], e, name[PLC])
5:      end for
6:      DrSet_CT ← DrSet_CT ∪ GENDR-PD(PD, PLC, TLC)
7:      return DrSet_CT
8:  end function
```

Listing 3: DR Generation for one process definition

6.3 Generation of General Revocation Triggers

In this section, we present a function to generate general revocation targets (GRT) which remove those cache entries during runtime which are not needed any more (cf. Section 5.3.3). Cache entries are revoked if either a task instance or process instance transitions into a final state. Hence, GRTs contain the information on which events an instance transits into an end state.

```
1:  function GENERATEGRT(PD, TLC, PLC)
2:      GrtSet ← ∅
3:      for all s ∈ F(PLC) do
4:          for all i ∈ in[s] do
5:              GrtSet ← GrtSet ∪ {GRT(i, name[PD])}
6:          end for
7:      end for
8:      for all t ∈ Q(PD) do
9:          for all s ∈ F(TLC) do
10:             for all i ∈ in[s] do
11:                 GrtSet ← GrtSet ∪ {GRT(i, name[t])}
12:             end for
13:         end for
14:     end for
15:     return GrtSet
16: end function
```

Listing 4: Generating general Revocation Targets

The algorithm GENERATEGRT (see Listing 4) generates the respective set of GRTs for a process definition (PD), and respective process and task life cycles (*PLC* and *TLC*). Both the runtime and the output size of this operation depend on the number of transitions to the end states of PLC and TLC and the number of tasks within PD:

$$|GrtSet| \leq \sum_{s \in F(PLC)} |in[s]| + |Q(PD)| \cdot \sum_{s \in F(TLC)} |in[s]|$$

where F(*PLC*) and F(*TLC*) denote the set of end states of a respective *LC*.

The algorithm is called for each process definition; for example, the process "Travel Request" (cf. Figure 1-1) and the presented life cycles of JBoss jBPM (cf. Figure 2-7 and Figure 2-9) result in the list of GRTs given with Table 18.

$GrtSet_{TravelRequest}$	*{ GRT("cancelProcess$_{PLC}$", "Travel Request"),* *GRT("endProcess$_{PLC}$", "Travel Request"),* *GRT("cancelTask$_{TLC}$", "Create Travel Request"),* *GRT("endTask$_{TLC}$", "Create Travel Request"),* *GRT("cancelTask$_{TLC}$", "Manager Approval"),* *GRT("endTask$_{TLC}$", "Manager Approval"),* *GRT("cancelTask$_{TLC}$", "Budget Approval"),* *GRT("endTask$_{TLC}$", "Budget Approval") }*

Table 18: General Revocation Targets (GRT) for the "Travel Request" process given in Chapter 1

6.4 Business Objects

In this section we introduce an algorithm to generate DRs, given the relations between business objects (BO) and tasks. Recall, function calls on BOs (BO-calls) also require access control checks. Access control checks are required for the user assigned to the task. We pre-evaluate access control requests as soon as a user claimed a task. For JBoss jBPM, "assign" is the event which assigns a user to a task and, hence, triggers the pre-evaluation. We further update the cache entry if the user gets revoked, i.e., the "revoke"-event occurs.

We assume that the relation between a BO and a task is defined with the process definition (PD). If the process definition language does not allow to define such relations, the security policy (SP) is another information source, as it should be defined for each BO which task a user must have claimed to be able to access the BO's functions.

Our algorithm (see Listing 5) retrieves the information about BO and task relations by calling GETBORELATIONS(*PD*), where PD is the process definition. This function is assumed to return a set of one-to-one relations between a BO and a task of the given process definition. One relation contains the information about the name of the BO, the name of the task, as well as all functions the task might call during its execution.

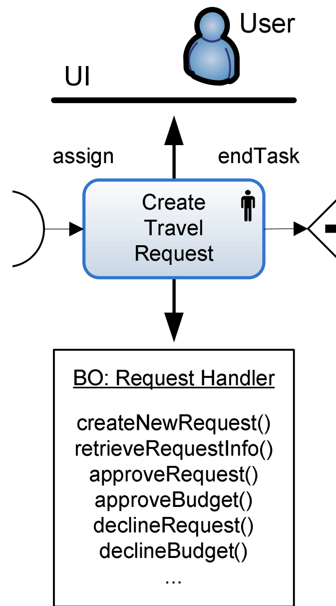


Figure 6-1: Task-to-BO relation for the task "Create Travel Request" given the "Travel Request" process introduced in Chapter 1

The algorithm GENBODR (see Listing 5) takes two arguments: the process definition and a set of events based on which pre-evaluations for access control requests should be triggered.

1: **function** GENBODR($PD, \{tEvents\}$)
2: $DrSet_{BO} \leftarrow \varnothing$
3: $relations \leftarrow$ GETBORELATIONS(PD)
4: **for all** $c \in relations$ **do**
5: **for all** $f \in BO\text{-}calls[c]$ **do**
6: **for all** $e \in tEvents$ **do**
7: $entry \leftarrow \mathrm{TT}(e, task[c], '*') \mapsto \mathrm{SR}(f, BO\text{-}name[c], 'event.\mathrm{user}', 'event.\mathrm{I.piid}')$
8: $DrSet_{BO} \leftarrow DrSet_{BO} \cup entry$
9: **end for**
10: **end for**
11: **end for**
12: **return** $DrSet_{BO}$
13: **end function**

Listing 5: Generation of DRs for task-to-BO relations

For our example process "Travel Request", we assume the task "Create Travel Request" accesses an BO called "Request Handler", offering the functions as depicted in Figure 6-1. We assume the task "Create Travel Request" only uses two of the displayed functions, namely "createNewRequest", and "retrieveRequestInfo". We call the algorithm with

GENBODR(PD, {"assign", "revoke"}), where PD is the process definition of the "Travel Request" process, "assign" and "revoke" the two events which trigger pre-evaluations. The respective DRs for this task are listed with Table 19.

$DrSet_{BO\text{-}CreateTravelRequest}$	*{TT("assign$_{TLC}$", "Create Travel Request", "*") ↦* *SR("createNewRequest", "Request Handler", event.user, event.*I.piid*),* *TT("revoke$_{TLC}$", "Create Travel Request", "*") ↦* *SR("createNewRequest", "Request Handler", event.user, event.*I.piid*),* *TT("assign$_{TLC}$", "Create Travel Request", "*") ↦* *SR("retrieveRequestInfo", "Request Handler", event.user, event.*I.piid*),* *TT("revoke$_{TLC}$", "Create Travel Request", "*") ↦* *SR("retrieveRequestInfo", "Request Handler", event.user, event.*I.piid*)}*

Table 19: Set of DRs for the task "Create Travel Request" given the "Travel Request" process introduced in Chapter 1

Both, runtime and output size of this algorithm depend on the number of relations, triggering events, and BO-calls per relation:

$$|DrSet_{BO}| \leq |tEvents| \cdot \sum_{c \in relations} |\text{BO-calls}[c]| \; .$$

6.5 Dynamic Access Control

In Section 5.4.1, we introduce three different definitions of constraint-specific DRs, each for one change-detectable context constraint (DSoD, BoD, and Dynamic Cardinality). All of them have the same general structure: they comprise a set of system events upon which updates of cache entries have to be performed, the event for which access control decisions should be pre-evaluated, as well as a set of resources (i.e., affected tasks) which have to be considered for updating cache entries.

We demonstrate the automatic generation of these constraint-specific DRs on the example of DSoD-DR. We describe an algorithm (cf. Listing 6) which generates the dependency relations based on the two input parameters *SP* and *PD. SP* is the security policy. *PD* reflects the process definition for which the constraint-specific DRs should be created.

Recall that the DSoD-DR requires the events upon which context changes occur (i.e., a user assignment to an exclusive task). This is a type of information which is very BPMS-specific. For JBoss jBPM these events are "assign" and "revoke" as we already described in Section 5.4.1; this can, however, be very different with other BPMSs and cannot be inferred from any of the information sources. Hence, it must be made available manually. We therefore assume that a function GETCONTEXTCHANGINGEVENTS returns the respective set of events (Listing 6, line 3).

Moreover, the algorithm extracts the required DSoD constraints from the security policy by calling the function GETDSODCONSTRAINTS, passing the policy and the process definition as parameters (Listing 6, line 4). We assume that each returned constraint comprises the access control relevant event for which access control decisions should be pre-evaluated, and the set of exclusive tasks for which the constraint must be checked during a process execution.

For each returned DSoD constraint, the algorithm generates the respective DSoD-specific DRs (DSoD-DR) (Listing 6, lines 5-7).

```
1:  function GENDR-DSOD(SP, PD)
2:      DrSet_DSoD ← ∅
3:      ctxChangingEvents ← GETCONTEXTCHANGINGEVENTS( )
4:      constraints_DSoD ← GETDSODCONSTRAINTS(SP, PD)
5:      for all c ∈ constraints_DSoD do
6:          DrSet_DSoD ← DrSet_DSoD ∪ {DSoD-DR(ctxChangingEvents, acrEvent[c], exclTasks[c])}
7:      end for
8:      return DrSet_DSoD
9:  end function
```

Listing 6: DSoD-DR Generation for DSoD Constraints

Both the runtime and the output size of this operation depend on the number of DSoD constraints:

$$|DrSet_{DSoD}| = |constraints_{DSoD}| \ .$$

Calling the algorithm given with Listing 6 for the "Travel Request" process annotated with DSoD constraints (see Figure 3-1), the same DSoD-DRs are created we already presented with Table 11. For better readability, we re-print them with Table 20 unterhalb.

These constraint-specific dependency relations give all the information a cache management requires for managing cache entries based on DSoD constraints. For demonstration, they can be transformed into extended Dependency Relations (extDR) to define specific successor requests (SR) as well as specific revocation targets (RT). The transformation adds to the demonstration that the complete caching heuristic introduced in our work can be expressed with extDRs, such that the cache management component must only be able to interpret this one type of DR. We will exemplarily transform $DSoD\text{-}DR_3$ from the above given list into a set of corresponding extDRs next.

DSoD-DR$_1$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* *affectedTasks = {*Task*("Create Travel Request"),* Task*("Manager Approval")}*
DSoD-DR$_2$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* *affectedTasks = {*Task*("Create Travel Request"),* Task*("Budget Approval")}*
DSoD-DR$_3$	*contextChangingEvents = {"assign", "revoke"}* *acrEvent = {"assign"}* *affectedTasks = {*Task*("Manager Approval"),* Task*("Budget Approval")}*

Table 20: DSoD specific Dependency Relations for the Travel Request process

Recall, each extDR requires one TT, one SR, and one or more RTs. There is a direct mapping between the TT and the SR such that an event matching the TT triggers the pre-evaluation of the SR to generate a cache entry. Matching events for the RTs revoke the generated cache entry.

A DSoD-DR defines a set of context-changing events, an access control relevant event for which access control decisions have to be pre-evaluated, and a set of exclusive tasks. Note, context always changes if one of the context changing events for one of the exclusive tasks occurs. Hence, pre-evaluations for cache updates must (only) be triggered at combinations of context-changing events and exclusive tasks. For DSoD-DR$_3$, this results in the following list of TTs at which cache updates must be triggered:

- TT(*"assign"*, *"Manager Approval"*, *"*"*),
- TT(*"revoke"*, *"Manager Approval"*, *"*"*),
- TT(*"assign"*, *"Budget Approval"*, *"*"*), and
- TT(*"revoke"*, *"Budget Approval"*, *"*"*).

If an event matches one of the listed TTs, cached decisions for all *other* exclusive tasks of the DSoD-DR have to be updated. Table 21 gives an overview of which TTs trigger the pre-evaluation of which successor requests, given DSoD-DR$_3$ as example.

TT*("assign", "Manager Approval", "*")*	SR*("assign", "Budget Approval", event.*user*, event.*I.piid*)*
TT*("revoke", "Manager Approval", "*")*	SR*("assign", "Budget Approval", event.*user*, event.*I.piid*)*
TT*("assign", "Budget Approval", "*")*	SR*("assign", "Manager Approval", event.*user*, event.*I.piid*)*
TT*("revoke", "Budget Approval", "*")*	SR*("assign", "Manager Approval", event.*user*, event.*I.piid*)*

Table 21: Mapping between TTs and SRs based on DSoD-DR$_3$

Finally, it is important that pre-computed access control decisions are revoked if they become obsolete. Remember, RTs define events upon which cache entries are removed. RTs are linked to cache entries. All RTs defined within an extDR revoke exactly those cache entries which have been pre-evaluated based on the same extDR's SR.

We define the general revocation targets (GRT) as RTs such that the cache entries get revoked if the process or task instance terminates. Table 22 shows the links between the above mentioned SRs and respective RTs, given DSoD-DR_3 as example.

SR(*"assign", "Budget Approval", event.*user, *event.*I.piid)	RT(*"cancelTask", "Budget Approval"*), RT(*"endTask", "Budget Approval"*), RT(*"cancelProcess", "Travel Request"*), RT(*"endProcess", "Travel Request"*)
SR(*"assign", "Manager Approval", event.*user, *event.*I.piid)	RT(*"cancelTask", "Manager Approval"*), RT(*"endTask", "Manager Approval"*), RT(*"cancelProcess", "Travel Request"*), RT(*"endProcess", "Travel Request"*)

Table 22: Mapping between SRs and RTs based on DSoD-DR_3

Given Table 21 and Table 22, we define an extDR for each given TT and SR. The resulting extDRs are given with Table 23.

$extDR_{MA1}$	TT = (*"assign", "Manager Approval", "*"*) SR = (*"assign", "Budget Approval", event.*user, {*event.*I.piid}) RT = { RT(*"cancelTask", "Budget Approval"*), RT(*"endTask", "Budget Approval"*) RT(*"cancelProcess", "Travel Request"*), RT(*"endProcess", "Travel Request"*)}
$extDR_{MA2}$	TT = (*"revoke", "Manager Approval", "*"*) SR = (*"assign", "Budget Approval", event.*user, {*event.*I.piid}) RT = { RT(*"cancelTask", "Budget Approval"*), RT(*"endTask", "Budget Approval"*), RT(*"cancelProcess", "Travel Request"*), RT(*"endProcess", "Travel Request"*)}
$extDR_{BA1}$	TT = (*"assign", "Budget Approval", "*"*) SR = (*"assign", "Manager Approval", event.*user, {*event.*I.piid}) RT = { RT(*"cancelTask", "Budget Approval"*), RT(*"endTask", "Budget Approval"*) RT(*"cancelProcess", "Travel Request"*), RT(*"endProcess", "Travel Request"*)}
$extDR_{BA2}$	TT = (*"revoke", "Budget Approval", "*"*) SR = (*"assign", "Manager Approval", event.*user, {*event.*I.piid}) RT = { RT(*"cancelTask", "Manager Approval"*), RT(*"endTask", "Manager Approval"*) RT(*"cancelProcess", "Travel Request"*), RT(*"endProcess", "Travel Request"*)}

Table 23: Example for extDRs based on the context constraint specific DSoD-DR_3

In summary, the transformation first defines the TTs at which cache entries must be updated. This is the case for *all* context-changing events for *all* exclusive tasks. Secondly, SRs are defined for all exclusive tasks, defining the pre-evaluation of new cache entries, potentially updating previously evaluated entries. SRs are linked to RTs which revoke these pre-evaluated cache entries whenever the task or process instance finishes.

Finally, each defined TT is mapped to one or more SRs. The mapping of a TT to an SR is done in such a way that a TT triggers the updates for all *other* exclusive tasks (e.g., a TT with an "assign"-event for the task "Manager Approval" is mapped to an SR which updates the cache entries for the task "Budget Approval", and vice versa).

For the other constraint-specific DRs (i.e., BoD-DR, and DCard-DR), the transformation into extDRs are similar. The main difference for BoD-DR is that, compared to DSoD and Dynamic Cardinality, context changes affect all users for which access control decisions are pre-evaluated. Recall, for binding of duties (BoD) one user must perform all tasks within a set of bounded tasks. If one user claims one of these tasks, no other user may claim any of the remaining bounded tasks. Hence, the respective SRs must use the already known function "*pdp*.getUsers" to update affected cache entries for all users for all other bounded tasks.

6.6 Summary

In this chapter we presented how the caching heuristic required for our caching strategy can be generated automatically. We introduced the respective algorithms and gave respective examples based on the process engine JBoss jBPM [62].

Main information sources for the generation of the caching heuristic may be provided by the workflow system's specification (i.e., life cycles), business processes, and relations between tasks and business object. Additional information for generating dependency relations come from the security policy which results in relations which trigger updates of cached access control decisions that became invalid due to possible context changes.

In the following chapter we will provide two additional approaches for optimizing caching of access control decisions for business process-driven environments.

7 Aspects for Optimization

In this chapter we describe two modes of operation to apply ProActive Caching (PAC) an optimized way. The first aspect introduces the re-usage of already pre-evaluated access control decisions. We specifically explain to what extend such a re-use is possible and which details for cache queries have to be considered.

The second aspect describes a two-levelled caching architecture which allows "hybrid caching", hence, the combination of PAC with other caching strategies. This specifically allows to choose strategies in such a way that, if combined, drawbacks from each single strategy can be compensated.

7.1 Reusing the Cache for Access Control Pre-Evaluations

Each access control decision is specifically evaluated for one process instance. This binds all cached decisions to exactly one instance. This is enforced by the PIID stored with each cache entry (cf. Definition 7, page 80). In consequence, for each new process instance a separate set of cache entries is pre-computed and discarded after the instance is finished. The pre-evaluation of access control decisions, however, causes significant overhead.

In this section we present an optimization of our caching strategy for reducing the overhead by allowing some cache entries to be re-used by the cache management whenever it is going to trigger the pre-evaluation of access control requests. This means, in very specific cases, cache entries which are already stored in the cache are re-used and do not have to be pre-computed a second time.

We first have to analyse which cached access control decisions are suited to be re-used. Afterwards, we will show three small adaptations to be made on the cache management.

Two types of Access Control Decisions

Recall, all access control decisions are based on the evaluation of (role-based) permissions defined with the security policy. Hence, they are evaluated for a specific resource (i.e., process or task definition), a specific user, and a specific event on the resource. Additionally, permissions may be further restricted on change-detectable context constraints and open constraints.

Access control evaluations which are not based on any dynamic context constraint do obviously not depend on any context information; they are valid for all process and task instances, independent of the current system status. The reason is, the security policy (and,

hence, its permissions) is specified for a specific process definition, rather than for a particular process instance of it. In consequence, independent of the process or task instance, the access control decision for a particular user and event on a particular resource (i.e., process or task definition) will always be the same.

This also holds for access control evaluations which are solely based on open constraints (OC). We previously described that OCs are part of a cache entry and - only when the entry is eventually consumed - all contained OCs have to be evaluated. Hence, context information which is dependent on the system state (i.e., process instance specific) is only required when the entry is eventually used. The on-the-fly evaluation of these OCs makes the resulting access control decision dependent on the respective process instance; the cache entry itself only containing the OCs to be evaluated is and remains process instance independent.

There are access control evaluations which are at least partially based on change-detectable context constraints. Those access control decisions are specifically evaluated for one process or task instance; hence, results may vary between different instances.

In essence, we have two types of access control decisions:

1. Access control decisions which are pre-computed for a specific resource (i.e., process and task definition), a user, and an event, which are based on permissions and possibly take OCs into account. These decisions are valid for any instance of a respective process definition.
2. Access control decisions which are pre-computed for a specific resource (i.e., process and task definition), a user, and an event, which are based on permissions as well as on the evaluation of change-detectable context constraints. These decisions are only valid for one specific instance of a respective process definition.

This requires a distinction between cached decisions which are valid across instances and cached decisions which are instance specific.

Cross-instance cache entries

The solution is using the process instance identifier (PIID) available in every cache entry. Cache entries which are independent of a process instance contain an empty PIID, entries which are evaluated based on change-detectable context constraints contain the PIID of the instance for which they have been evaluated.

This requires an adaptation at the cache management component. The cache management component triggers pre-evaluations and stores the resulting access control decisions as cache entries. Hence, this component must be aware for which cache entries the PIID has to be set (i.e., instance specific evaluations) and for which cache entries the PIID must be left empty (i.e., are consumable across process instances).

The solution is to have dependency relations (DR) containing a flag, whether a PIID should be stored with a cache entry or not. Recall, the successor request (SR) is part of a DR and contains all information the cache management requires for triggering an access control request evaluation and storing its result with the cache. We extend the current definition of an SR and define a *cross-instance successor request* which contains the required information as Boolean flag.

Definition 19 Cross-instance Successor Request

A *cross-instance Successor Request* (ciSR) is a tuple ciSR(SR(*e*, *r*, *u*, *piid*), *ci*) where

- *SR* is a successor request (as defined with Definition 9), and
- *ci* is a Boolean flag which, if *true*, indicates that the resulting access control decision is valid across process instances and should be stored without PIID.

Each pre-evaluation of an SR that specifies an access control request relying on one or more change-detectable context constraints, results in an access control decision which is instance specific and, hence, not valid across process instances. For these SRs, the flag *ci* is set to *false*. Pre-evaluations for all other tasks are valid across instances; hence, for these SRs the flag *ci* is set to *true*.

Cache Queries

A second adaptation is with the cache management and its sequence of interactions with other components. The sequence diagram given with Figure 7-1 unterhalb shows the single interactions the cache management has to execute to perform all steps required for managing the cache upon receiving a new *trigger event*. Compared to the sequence diagram we presented previously (cf. Figure 5-10), the cache management additionally checks whether already cached entries can be re-used, instead of triggering another pre-evaluation (highlighted in blue in Figure 7-1). Only those SRs which cannot be satisfied by querying the cache are sent to the PDP for pre-evaluation.

Revocation of Cross-instance Cache Entries

A third adaptation requires the cache management to keep cross-instance cache entries as long in the cache as they are needed, and only revoke them if the last process instance for which they may potentially be used is finally terminated. The sequence diagram given with Figure 7-1 shows this aspect with the loop for revoking obsolete entries. For each entry triggered to be revoked, the cache management must check, whether the revocation is based on the last process instance which might use the cache entry, or the entry is instance specific (i.e., containing an PIID). Only if this is the case, the cached decision can be revoked.

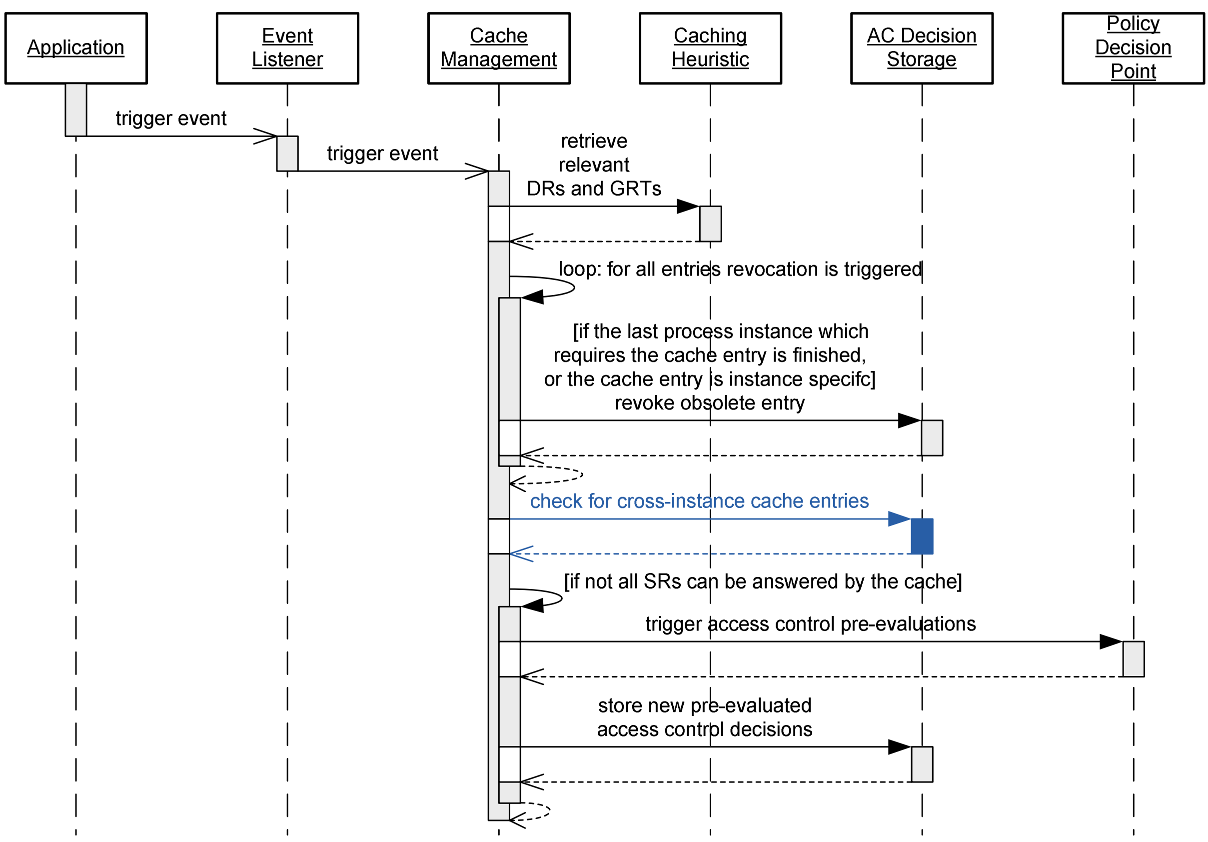


Figure 7-1: Sequence diagram for pre-evaluating cross-instance access control decisions

The cache management may handle which process instances are still active in the system either based on the events the process engine broadcasts (i.e., listening to the events "createProcess", "cancelProcess", and "endProcess"), or the BPMS provides this information as context provider.

Analysis

We stated that the pre-evaluation of access control decisions causes significant overhead. For investigating the time required for the pre-evaluations, we simulated the execution of different processes while with each iteration we increased the number of instances per process definition[5].

The goal of the simulation is to analyse the effort it requires to pre-evaluate access control decisions for the execution of multiple, in parallel executed process instances. Table 24 summarizes the results of the evaluation. For each iteration, we report the accumulated processing times (in seconds) required to process all pre-evaluations required to fill the

[5] For more information on our implementation and environment, please see the detailed performance analysis described in Chapter 8.

cache in accordance to our caching strategy. In fact, we report on the accumulated processing time if all entries are instance specific and no cache entry is re-used, as well as on the accumulated processing time if cache entries can be re-used as described in this section.

# of process instances	instance-specific pre-evaluation	cross-instance pre-evaluation	pre-evaluation speed up
25	772 s	105 s	7.4
50	1232 s	110 s	11.2
75	1654 s	116 s	14.3
100	2016 s	120 s	16.8
150	3145 s	132 s	23.8
200	4102 s	149 s	27.5
250	5006 s	151 s	33.2
300	6346 s	160 s	39.7

Table 24: Accumulated response times for instance-specific as well as cross-instance pre-evaluations (w/o DSoD constraints) of access control requests for multiple, in parallel executed process instances

The results clearly show that the optimization presented in this section significantly reduces the overhead required for pre-evaluating access control requests. In those cases were cross-instance pre-evaluation is used, the accumulated time for pre-evaluations is reduced by an increasing factor between 7 and almost 40. This increase can be explained by the fact that the more instances are running in parallel the more often can cache entries in the cache actually be re-used.

In summary, it is possible to use cached decisions for more than one process instance. While using this approach, it is important that there is a clear label for instance-specific access control decisions. It is important that they may only be used for the instance they have been evaluated for. This can easily be done by using the PIID as such a label. Additionally, DRs have to be defined along Definition 19 using a cross-instance Successor Request such that the cache management labels cache entries respectively.

In general, our brief performance analysis clearly shows that with this optimization the overhead for pre-evaluations can significantly be reduced, in some cases by a factor of almost 40. In particular, this shows that PAC is scalable, i.e., the effort required for pre-computation does not increase significantly with increasing number of active process instances.

7.2 Hybrid Caching Architecture (2-Level Cache)

In this section we describe a hybrid caching architecture introducing a two-levelled cache. The goal is allowing the combination of different caching strategies such that drawbacks from single caching strategies can more easily be compensated.

The obvious drawback from standard and related caching approaches (e.g., $SAAM_{RBAC}$ [68] introduced in Chapter 4) is the lack of the ability to cache access control decisions which are based on dynamic context information. Evaluating dynamic context constraints, however, is a major requirement for access control enforcement within business process-driven systems. The drawback of PAC is the required overhead for pre-evaluating access control decisions which we described in the previous section.

We propose a hybrid caching approach to combine caching strategies. The general architecture is depicted with Figure 7-2.

The complete cache is embedded in the system architecture in the same way PAC was added to our reference architecture illustrated in Figure 5-3; it functions as proxy in between the PEP and the PDP, receiving access control requests and returning access control decisions. If the cache does not contain an answer to a query, the access control request is transparently forwarded to the PDP for regular evaluation, and the PDP's answer returned to the PEP.

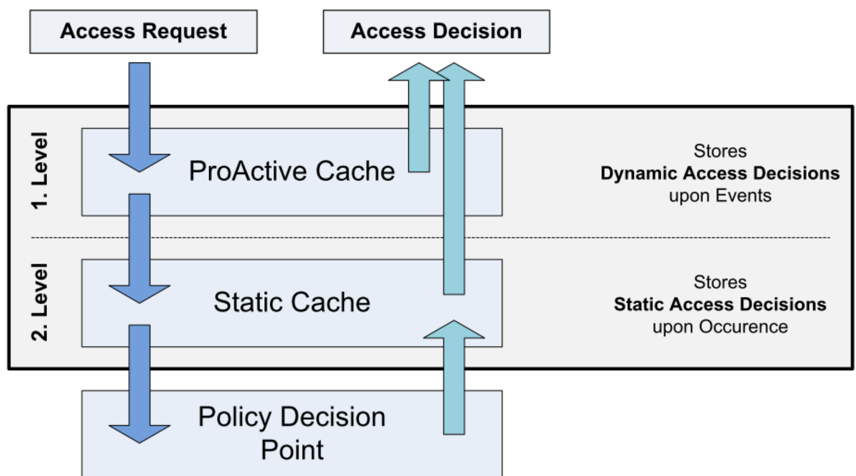


Figure 7-2: A multi-levelled caching architecture

The cache itself has two levels which sequentially process each incoming access control request. Level 1 of the cache incorporates PAC, level 2 any other standard caching approach which suites for caching access control decisions.

The PAC on Level 1 handles all access control requests which cannot be cached by any other caching strategy; hence, it is responsible for requests which require dynamic context

constraints for evaluation (e.g., DSoD, BoD, etc.). It therefore contains either cache entries which are specifically evaluated for one process instance or cache entries which contain open constraints.

In consequence, it is only required to define dependency relations (DR) for exactly those tasks and events, for which dynamic context constraints are to be considered. For our process "Travel Request", for instance, only the constraint-specific DRs of Table 20 (page 122) are required. Every time an event triggers one of these DRs, pre-evaluations are performed and new cache entries are inserted into the cache on Level 1 (illustrated with Figure 7-3); similar, cache entries are revoked in a respective way.

Similar to our approach for using cross-instance cache entries described in the last section, cache entries which are specifically evaluated for one process instance contain the PIID of the respective instance such that they can be unambiguously related to that specific instance.

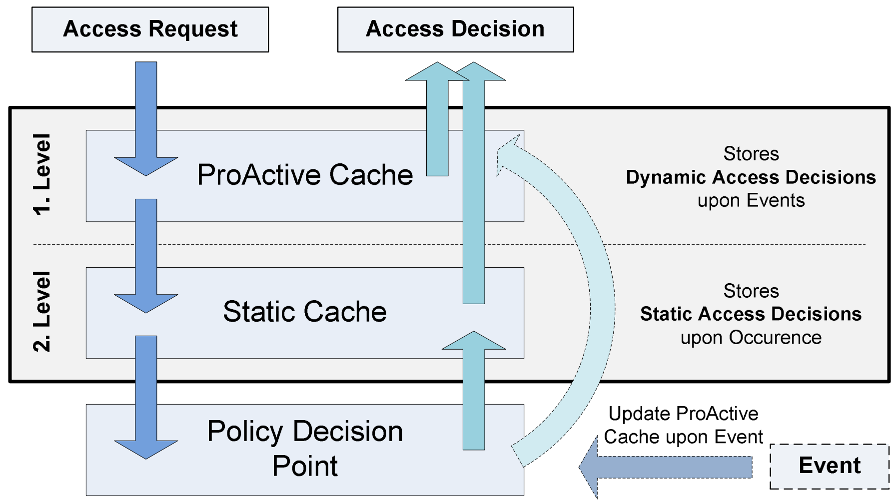


Figure 7-3: Multi-level caching architecture - cache update illustration

All access control requests which do not fall under PAC's responsibility are transparently forwarded to the cache on level 2. A cache hit on level 2 leads to an immediate response. The cache on level 2 applies its own specific caching strategy; hence, $SAAM_{RBAC}$ as level 2 caching strategy would return approximate decisions using the current content of the cache as we described in Chapter 4. A cache miss leads to a regular access control evaluation by the PDP. The result is first returned to the cache on level 2 such that the caching strategy it implements may incorporate the result for future queries before it is eventually returned to the PEP for access control enforcement.

The cache on the second level is responsible for storing access control decisions which are not restricted by dynamic context constraints. The respective strategy might be standard

caching, i.e., storing access control decisions whenever they are returned by the PDP, $SAAM_{RBAC}$, or any other strategy available for caching access control decisions. In case of $SAAM_{RBAC}$, of course, an access control model based on RBAC is mandatory.

We conducted performance analysis for different caching strategies, namely standard caching, $SAAM_{RBAC}$, PAC, as well as hybrid combinations. We report on the results in the following Chapter 8.

7.3 Summary

In this chapter we introduced two modes of operation to run ProActive Caching (PAC) in an optimized way. We first introduced cross-instance caching which enables the re-use of cached access control decisions across process instances. In this respect, it is important to distinguish between access control decisions which are specifically evaluated for one process instance, and access control decisions which solely rely on open constraints or no constraints at all. We showed that only the latter can be consumed across instances and introduced means to enable a distinction.

Moreover, we introduced a two-levelled caching hierarchy which enables the combination of PAC with other caching heuristics. In the following Chapter 8 we will analyse and compare different caching strategies, including this two-levelled caching architecture.

8 Analysis

Caching access control decisions is a common way of increasing the overall system performance and, thus, minimizing the delays that users observe. Our ProActive Caching strategy is to our knowledge the first workflow specific caching strategy which exploits the business process and workflow models for pre-evaluating access control requests and avoiding cache misses. We applied and analysed ProActive Caching for improving the performance on static and dynamic access control evaluations.

In this chapter, we present our analysis of ProActive Caching according to performance improvements and provide a detailed comparison with other approaches for caching role-based and dynamic access control decisions in business process-driven systems. In particular, we compare standard caching against caching strategies developed to specifically cache access control decisions, i.e., $SAAM_{RBAC}$ and ProActive Caching.

We start with briefly introducing the caching approaches we considered for our analysis in Section 8.1 and our test environment in Section 8.2. Our performance results are presented in Section 8.3 for static environments (i.e., where no dynamic context constraints are applied) and in Section 8.4 for dynamic environments. Each of the later two sections concludes with a discussion of the observed results.

8.1 Functional Classification of Caching Approaches

For our performance analysis, we consider the following caching approaches for caching access control decisions in business process-driven systems:

1. *Standard caching* (SC). In a standard cache, each access control decision which is not found in the cache is evaluated by the PDP and its response stored in the cache, optimizing identical access control requests in the future. As SC was developed as a generic caching strategy, it neither exploits specific features of the access control model used, nor of any underlying process models.

2. *Secondary and approximate authorization model* (SAAM), in particular, $SAAM_{RBAC}$ [68]. SAAM[6] is optimized for caching decisions for RBAC models. SAAM uses already cached access decisions to infer further access decisions which have not yet been cached. Inferred decisions are called approximate decisions. Hence, also new access control requests which did not appear yet can possibly be answered by a

[6] In this chapter, we only refer to $SAAM_{RBAC}$ such that we neglect the subscript and write SAAM for better readability.

cache look-up. SAAM exploits specifics of the RBAC access control model, but is not tailored for usage in business process-driven systems and, in particular, does not support caching of access control decisions based on dynamic context constraints. We introduced SAAM in Chapter 4.

3. *ProActive Caching* (PAC), our caching strategy especially developed with both, process-driven environments and dynamic security requirements in mind.

4. *Hybrid Caching* (hybrid SC and hybrid SAAM), a combination of PAC and a second caching strategy as introduced in Chapter 7. We use SC and SAAM as caching strategies for level 2 of the cache.

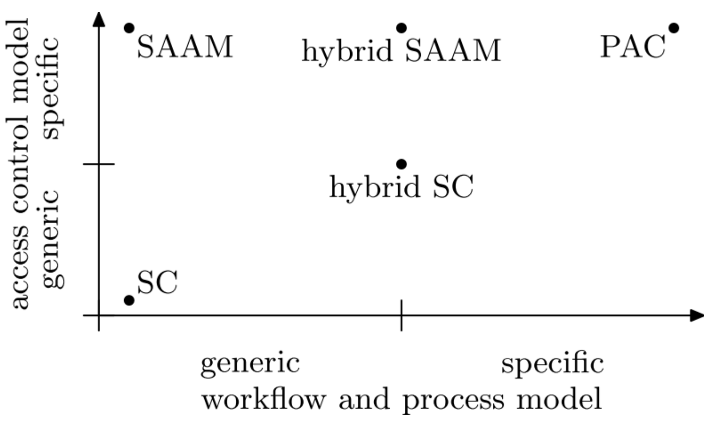


Figure 8-1: Taxonomy of caching approaches

The above mentioned caching approaches can be categorized with respect to their orientation towards business process-driven systems into:

1. strategies that are optimized for a specific caching domain (e.g., caching of access control decisions) versus strategies that are designed for caching arbitrary data, and
2. strategies that are optimized for workflow management systems (i.e., inherently exploiting process or workflow models) versus strategies that are designed for arbitrary systems.

Figure 8-1 illustrates the categorization for the given caching strategies.

8.2 Environmental Setup

We compare the different caching strategies based on our GWL scenario motivated in Section 2.2. We simulate a process-driven system and measure the time spent in evaluating access control requests that are required for displaying a user's GWL. In more detail, we use

an implementation of the architecture discussed in Section 5.2.2. We configured the system with various process models, taken from the set of reference models for SAP's R/3 system [16, 37]. (These process models are defined as Event-driven Process Chains (EPCs); example processes as well as a straight forward example for transforming them into BPMN [45] can be found in the Appendix). Notably, we enriched these process models with a role-based security policy and, depending on the experiment, with DSoD constraints.

In our experiment, we simulate the user-driven execution of several instances of the different business processes in parallel and measure the time for evaluating all access control requests that are required for displaying the GWL. If we look on a single process execution in isolation, our experiment resembles a scenario in which a user calls his GWL, selects (and claims) a task to work on and successfully finishes this task. Thereafter, he works on the upcoming tasks until the process is finished. Branches within a process are considered according to its structural element; in cases of multiple options for further path executions (i.e., OR and XOR gates), paths are randomly chosen. Considering such a workflow execution for many process instances allows for comparing the caching strategies for different workloads of the system.

In total, we consider process definitions of different sizes, selected from SAP's R/3 reference system [16, 37]. Each of the process definitions has between 4 and 25 tasks such that we divide them into groups of small (4 tasks), medium (7 tasks), and large (up to 25 tasks). We consider scenarios ranging from 25 to 300 process instances executed in parallel. Each execution is based on four process definitions out of one group, for which up to 300 process instances are created.

All process definitions are of non-linear type such that different paths of execution are possible. For each execution of a process instance, the path is randomly chosen. Hence, although process definitions have up to 25 tasks, the number of actually performed tasks per instance varies.

Our analysis shows that the selection of process definitions according to their size (i.e., small, medium, and large) only influences the total time of execution; the significant characteristics of the caching strategies we compare, however, remain the same. Hence, in the following sections we report on our analysis of the group for medium process definitions as we consider them for most significant in companies. With 300 process instances this results in the execution of about 1200 tasks. The results for both other groups differ, for instance, in the amount of time it takes to pre-evaluate all access control decisions to be placed into the cache, but are similar in their general characteristic we describe below.

The setting for our security policy comprises 100 users, 20 roles, and about 8000 permissions. Every user is a member of five randomly chosen roles. Each process is assigned to two roles that are allowed to instantiate and execute an instance of it. For performance tests with DSoD constraints we enriched our process definitions with DSoD constraints such that in average 40% of the tasks are affected by these constraints.

For all experiments we report on, we used a standard server with Intel Xeon CPU, 3.4 GHz, and 8 GByte RAM. The benchmark environment and all caching strategies are implemented with Java 1.6.0 Update 16.

PAC is implemented in its optimized version where cross-instance cache entries are available (cf. Chapter 7). Moreover, PAC is assumed to be able to pre-evaluate and cache almost all required access control decisions for the execution of our sample processes such that we only expect a minimal amount of cache misses. Cache misses will occur as we do not pre-evaluate access control decisions for the creation of a process instance, i.e., for the event "createProcess" (cf. Section 6.1). SC is also implemented such that cache entries can be consumed independent of the current process instance. SAAM is implemented in accordance to the algorithms presented in [68].

In the following, we first have a look on access control requests in a static environment (i.e., no dynamic context constraints are applied). In particular, we compare scenarios with no caching, SC, SAAM, and PAC. Afterwards, we add scenarios within a dynamic environment, comparing no cache, SC, SAAM, PAC, and hybrid caching.

8.3 Static Access Control

The number of access control requests that need to be evaluated for displaying a user’s worklist depends on the system state and the number of active tasks. The main criterion for displaying an active task in a user's GWL is that the user is allowed to claim and execute it. The execution of 300 process instances of medium size (i.e., 7 tasks) in parallel results in about 1200 worklist calls, each of them triggering multiple access control checks for all active and assignable tasks in the system. Checks are made for the user calling the worklist to check which of them may possibly claimed by the user.

Figure 8-2 unterhalb, for example, depicts on the horizontal axis all 1200 worklist calls for the execution of a scenario with 300 process instances using SC. For each call, the number of access control requests is shown, scaling according to the graph’s right vertical axis; additionally, for each call the time required to perform all access control requests and display the worklist is shown, scaling to the graph’s left vertical axis. As a first observation, this makes evident that caching profits from large number of processes being active: the more processes are running in parallel the more access control requests are evaluated to

display a worklist (the fine line slightly crosses the mark of 200 access control requests). Still, as towards the end of our scenario most requests are already cached, the time required for access control responses decreases.

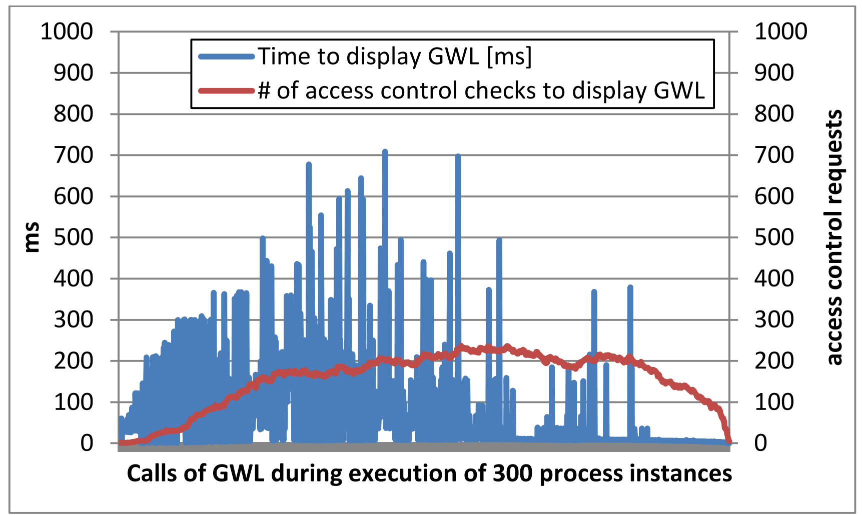


Figure 8-2: Displaying all worklists during one scenario execution (SC, 300 processes)

Figure 8-3 to Figure 8-6 summarize the comparison of executing our GWL-scenario using no caching, SC, SAAM, and PAC by relying on static access control policies. Static means that we do not consider any dynamic context constraints for evaluation. Furthermore, we assume that the policy remains unchanged.

The average time required for evaluating a single access control request without caching (depicted with Figure 8-3) slightly increases with the number of process instances running in parallel. PAC, in contrary, is independent hereof. Moreover, PAC results in a much faster evaluation of access control requests, always remaining faster than 1 ms (while 'no caching' results in evaluation times between 27 ms and 38 ms). A similar behaviour is observed for the average time required for displaying the complete GWL (displayed with Figure 8-4).

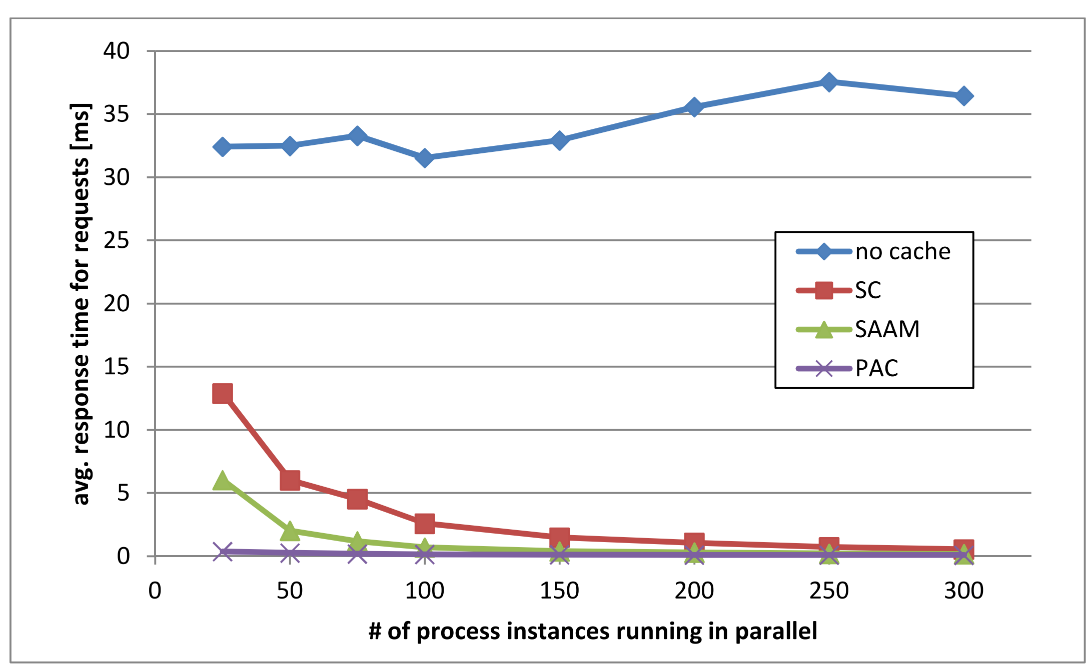


Figure 8-3: Static access control - average access time

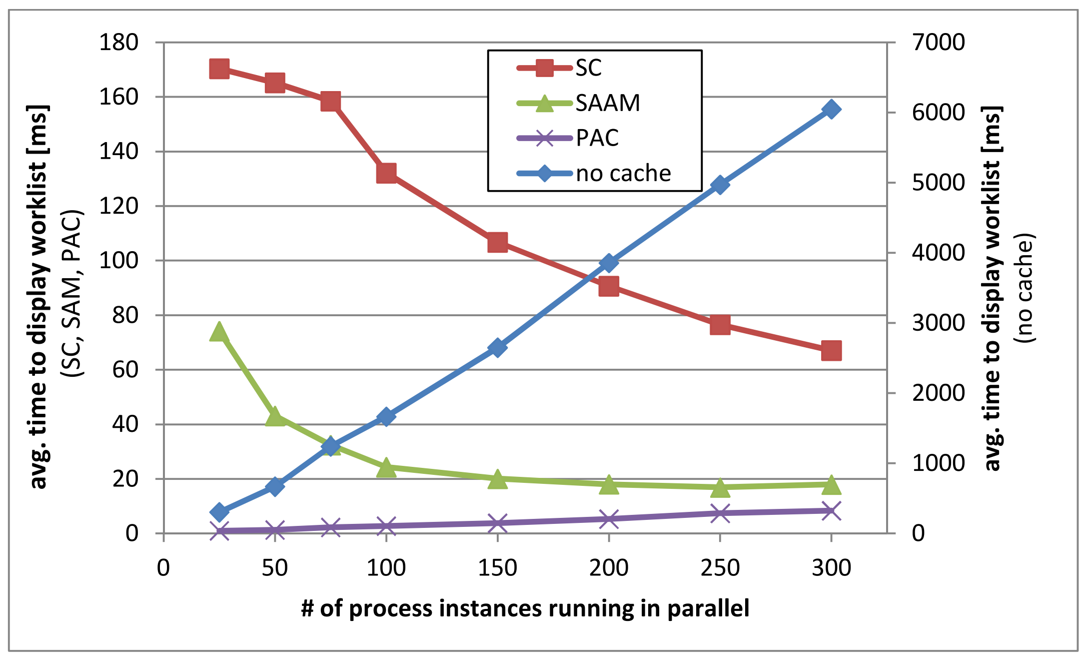


Figure 8-4: Static access control - average time for worklist display

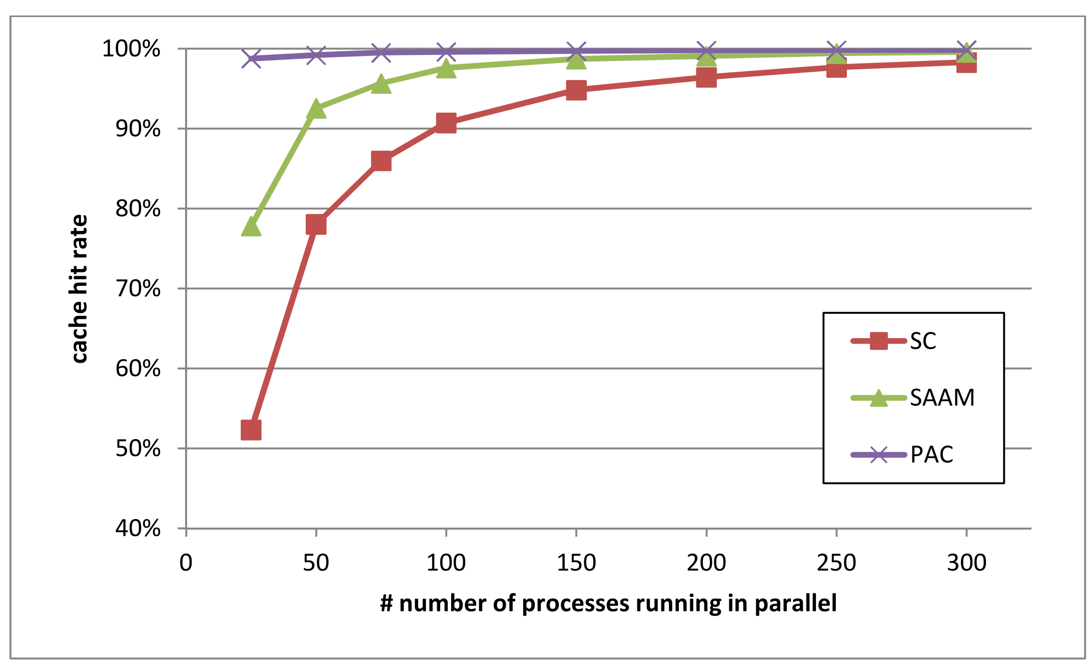


Figure 8-5: Static access control - hit rate

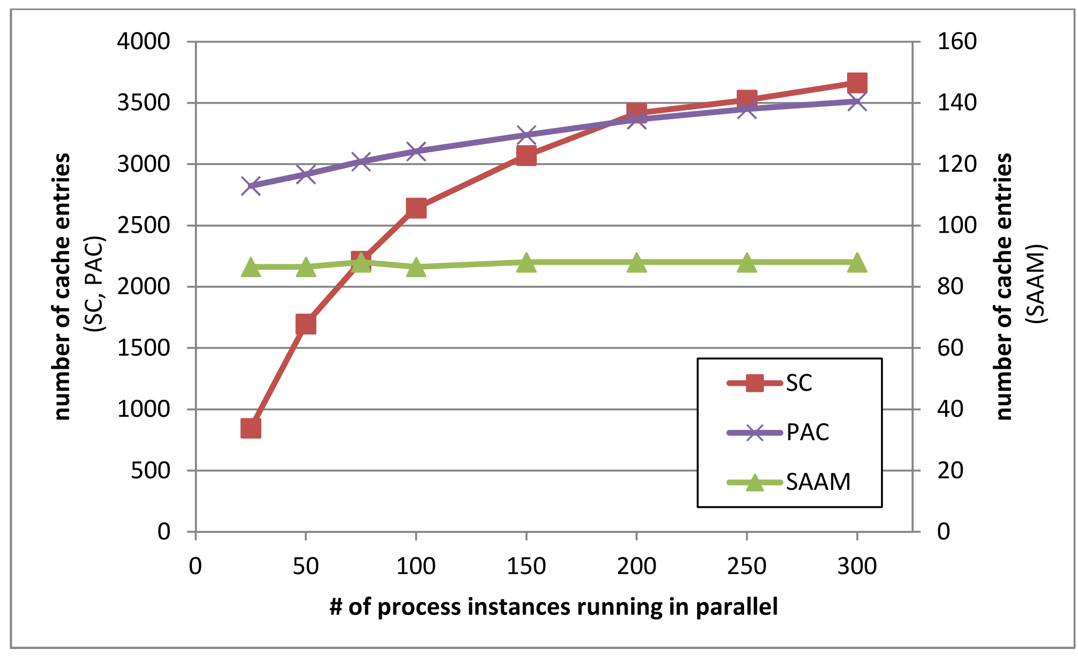


Figure 8-6: Static access control - maximum cache size

With an increasing number of parallel process instances, both SC and SAAM converge to PAC's low response time. This behaviour is due to the fact that with more processes running in parallel, the likelihood that an access control decision is already cached increases. This assumption is verified by the hit rate of the different caching strategies (see Figure 8-5) which converges to 100% for SAAM and SC. Our experiments support the results of Wei et al. [68]: in comparison to SC, SAAM converges significantly faster to the response time of PAC.

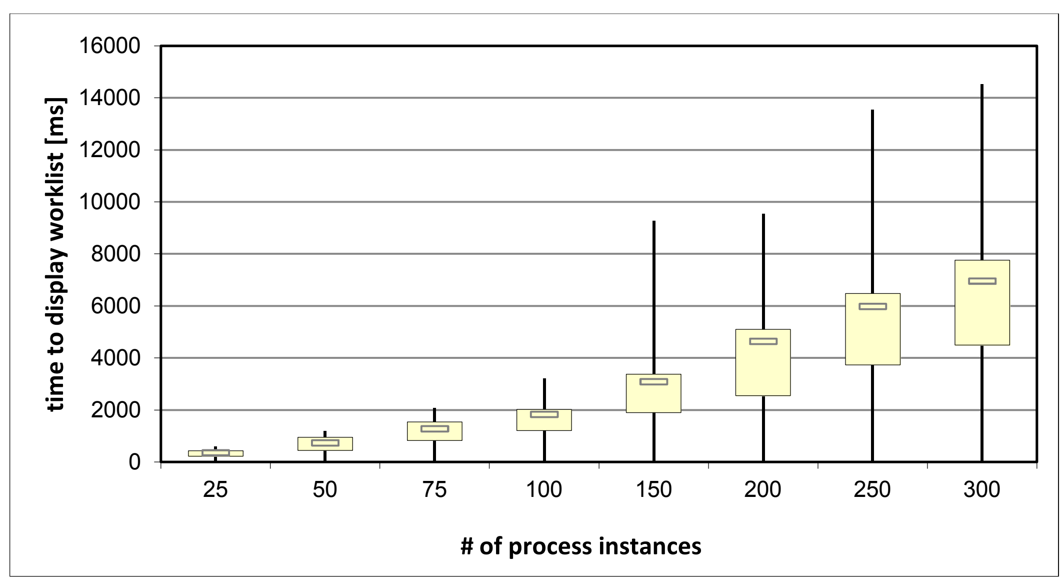


Figure 8-7: Static access control - GWL No Cache

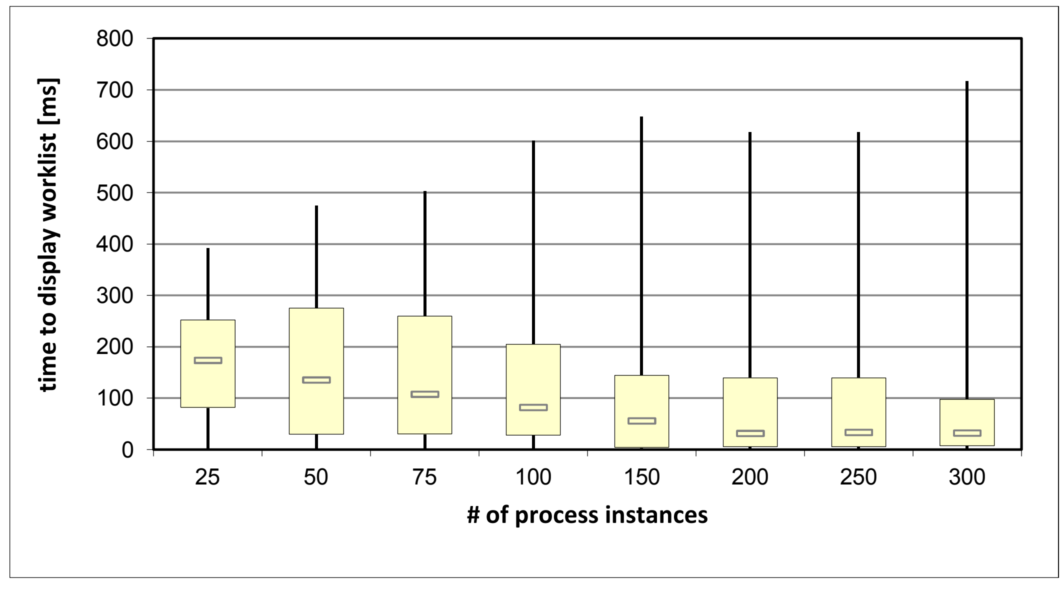


Figure 8-8: Static access control - GWL SC

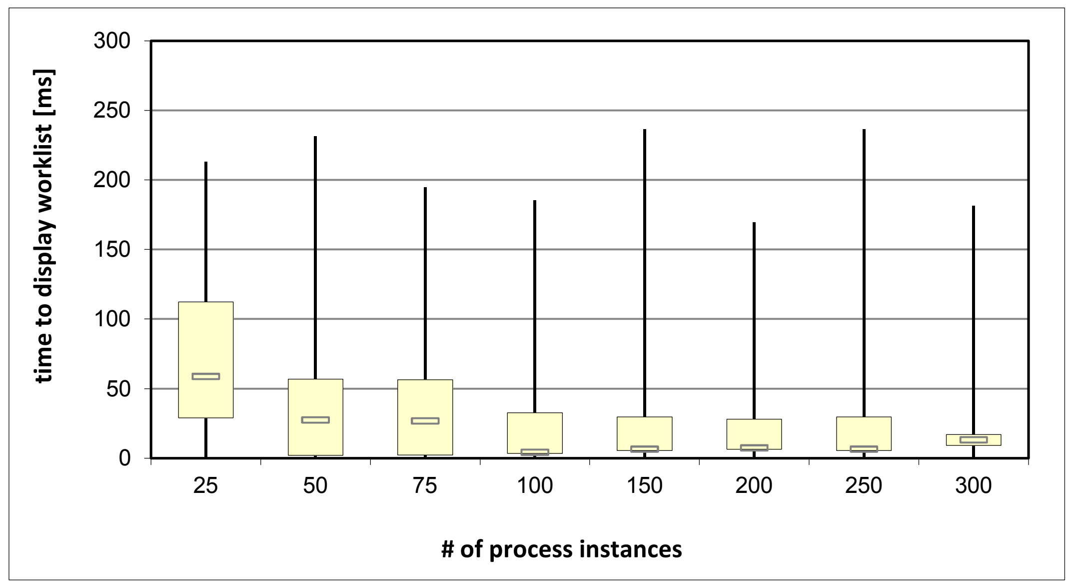


Figure 8-9: Static access control - GWL SAAM

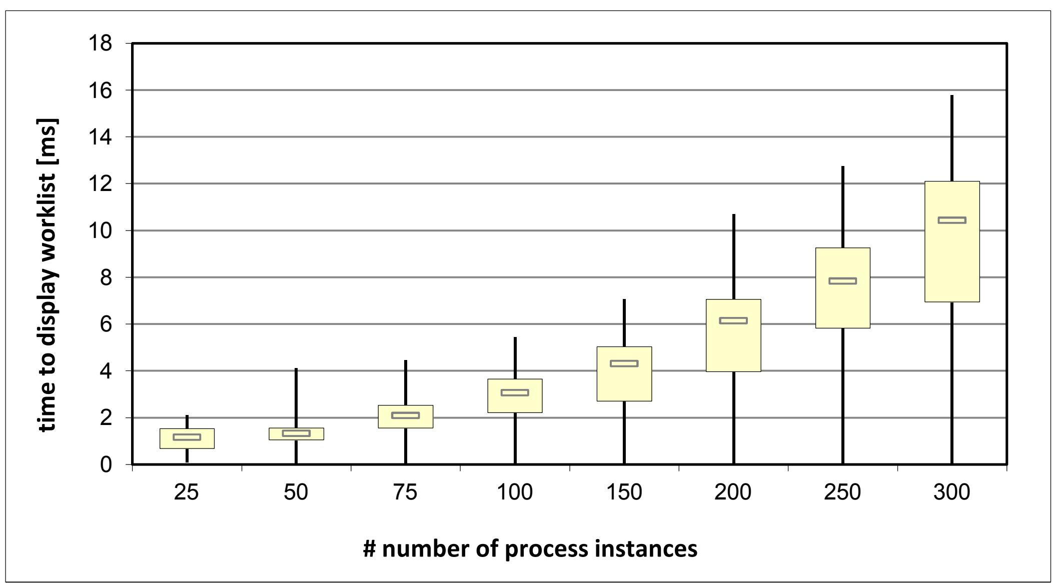


Figure 8-10: Static access control - GWL PAC

Figure 8-7 to Figure 8-10 illustrate the different times it takes to display the GWL using *box-plots*: the heights of the fine lines above and below the boxes show the minimal and maximal time required to display a worklist. The boxes start and end with the lower and upper quartiles of the required times. Additionally, the graphs show the experiments' median values, illustrating the maximum time a user has to wait in 50% of all cases. Note: the scaling of all box-plots is respectively adjusted to the results the graphs depict and varies among the graphs.

Displaying the worklist using no caching (see Figure 8-7) takes for 300 process instances up to 8 s in 75% of all cases. As the median is very close to the upper box limit, we conclude that only a few cases take significantly longer. In fact, our data shows that in up to 99% of all cases the user has to wait up to 8.5 s.

For SC (see Figure 8-8) a user waits up to 700 ms for the worklist to display. The probability that a user has to wait more than 100 ms, however, decreases with increasing number of process instances. This is already substantially faster compared to the implementation without caching (see Figure 8-7) where delays in the range of seconds are normal for the average case.

For SAAM (see Figure 8-9) the maximal time is lower as for SC and does not exceed 250 ms. Especially the cases for 300 process instances are very dense with an upper box limit at about 17 ms to display a worklist. In fact, our collected data shows that in 93% of all cases the required time remains below 50 ms.

The time for PAC (see Figure 8-10) are always less than 16 ms. Comparing the tests, starting from 25 up to 300 process instances, the average values to display the worklist even increase. This is due to PAC's caching strategy which results in a nearly 100% cache hit rate (cf. Figure 8-5). Hence, nearly all access control requests required to display a worklist are answered by the cache (rather than by the PDP) such that the time to display the worklist is the sum of all cache lookups - which obviously increases with an increasing number of cache lookups.

Small and Large Process Definitions

We conducted our analysis of business process executions for different sizes of business process definitions. We already described above that the general characteristic for executions of process definitions having different sizes remain similar, such that we reported on results using process definitions of medium size.

For the sake of completeness, below we depict the graphs for average response times for single access control requests, given small and large process definitions as input. The setup for the test with small business process definitions (4 tasks) as well as large process definitions (having up to 25 tasks) corresponds to the one used for the medium sized processes we reported on in Section 8.2.

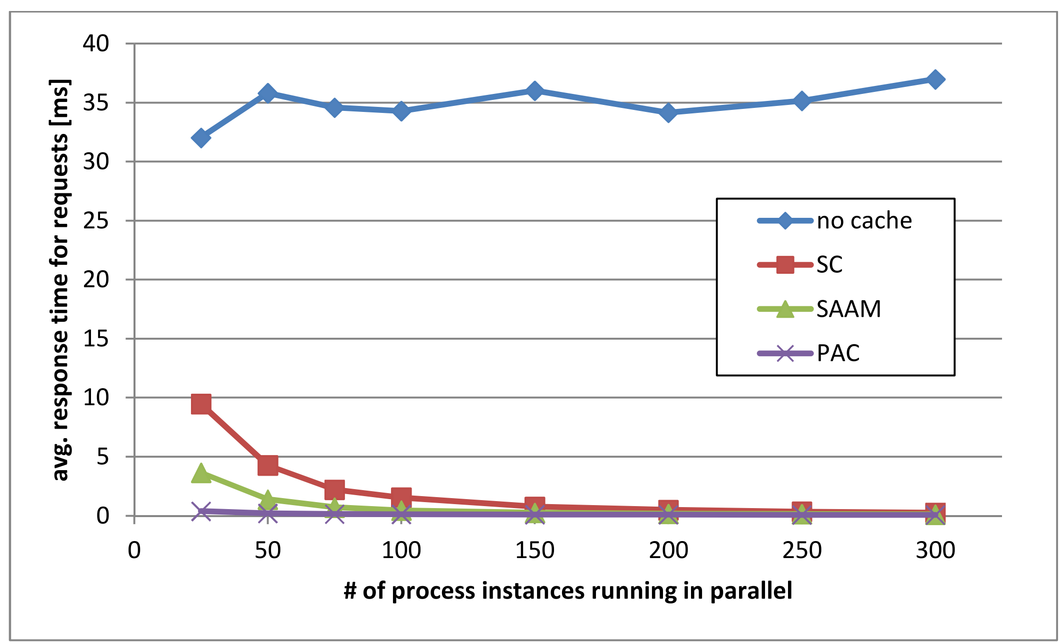


Figure 8-11: Static access control - average access time for small business process definitions

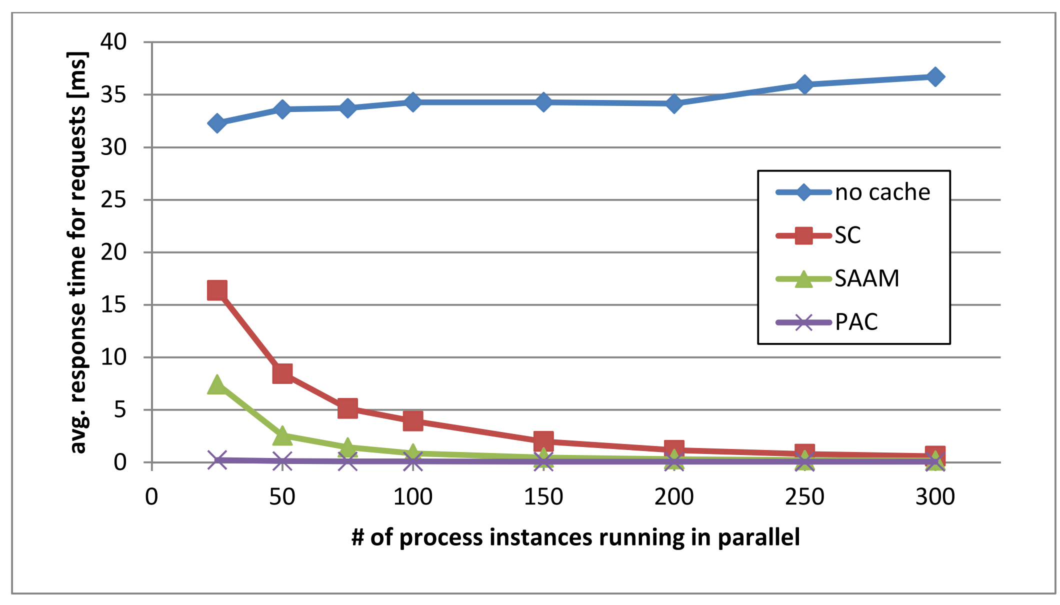


Figure 8-12: Static access control - average access time for large business process definitions

Discussion

The results show that for static environments caching is very effective in reducing the response time for access requests. Notably, even the standard cache for small scenarios already decreases the average access time to half of the time required in implementations without cache.

Further, in our experience both caching strategies developed for static environments (SC and SAAM) are easy to implement and achieve, already with a comparatively small amount of process instances, a benefit comparable to the PAC. Figure 8-2 shows that only in the case the cache is empty both strategies require a warm-up phase. During this warm-up phase, the response time of the system may be considerably higher. The maximum values in Figure 8-7 to Figure 8-10 confirm this observation.

While the PAC implementation requires a higher effort, it results in a system which can provide very low response times. They remain below a time of, in our case, 16 ms to deliver all access control decisions required for displaying a GWL. This is due to the fact, that PAC does not have a warm-up phase (as it pre-evaluates access requests) and thus, the deviation of times needed for evaluation is small. Albeit, we need to invest additional system resources for pre-computing cache entries that might never be used.

Furthermore, caching comes at a cost: for all caching variants we have to store the cached access control decisions in memory. Figure 8-6 illustrates the maximal number of cache entries for SC and PAC (left y-axis) and SAAM (right y-axis). The graph shows that already for small scenarios, the heuristics used by PAC result in pre-evaluating and caching nearly all access control decisions beforehand. In contrast, SAAM requires a much smaller number of cache entries. One has to take into account, however, that this comparison ignores the fact that the cache entries of SAAM are typically larger than the cache entries of PAC and SC. Measuring the memory consumption reveals that SAAM requires approximately half the memory as PAC or SC for large scenarios.

In conclusion, all caching strategies tested generate a strong impact with respect to a system's response time compared to systems without caching. If a system requires a guaranteed response time PAC seems to be most suited and the additional effort required for PAC seems to pay off. Moreover, this already proves the additional benefits of exploiting process and workflow models for caching access control decisions on static system resources.

8.4 Dynamic Access Control

In the following, we enrich our scenario with dynamic access control requirements, i.e., policies with dynamic context constraints. For our analysis, we use DSoD constraints as

prominent example of dynamic context constraints. In overall, we prepared 40% of the tasks of a process definition to be affected by a DSoD constraint. DSoD constraints rely on dynamic context information. A cache storing access decisions which are based on the evaluation of DSoD constraints, hence, needs to be updated if the context changes. PAC supports such cache updates while SC and SAAM do not.

Applying SC and SAAM in a dynamic environment requires that no access control decisions based on the evaluation of DSoD constraints are cached. Hence, we specifically forward all access control requests which require DSoD constraints to be evaluated directly to the PDP for regular evaluation. The information which requests have to be forwarded is available as it is clear which tasks of the used process definitions are affected by a DSoD constraint.

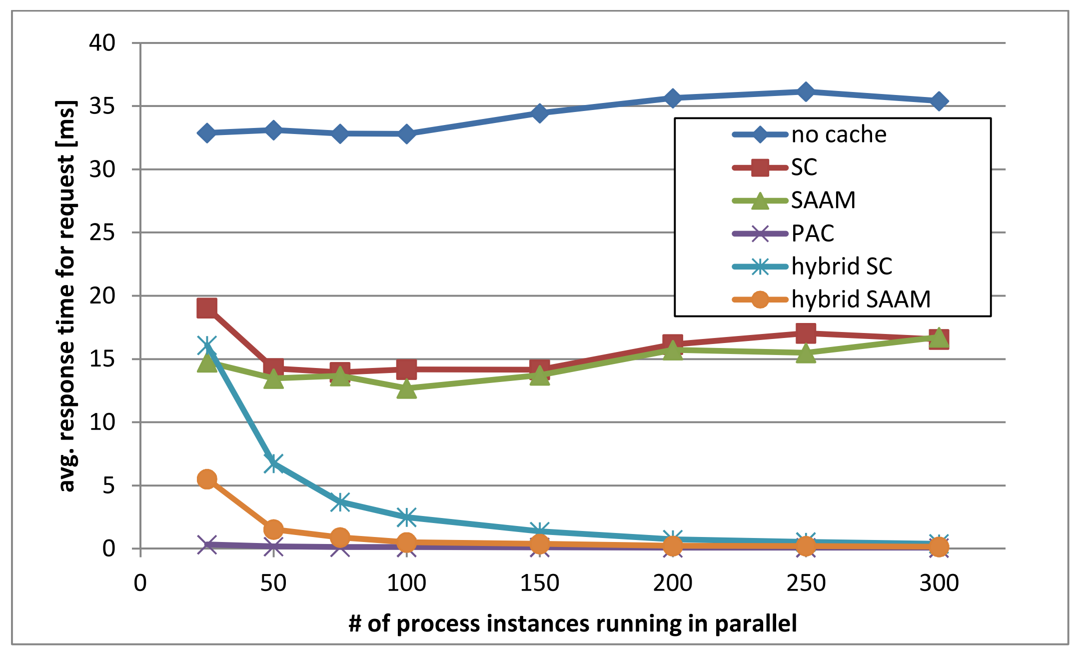


Figure 8-13: Dynamic access control (DSoD) - average access time

Figure 8-13 shows the average time it takes for each caching strategy to evaluate an access request. Similar to the static scenario, the response time for no caching increases with the number of process instances. SC and SAAM tend to stabilize in our experiments at a level around 16 ms. The reason for this behaviour is that dynamic access control decisions are not cached and, hence, every dynamic access request must be evaluated by the PDP. The cache hit rate of SC and SAAM (see Figure 8-14) roughly converges against a value equal to the probability that an access request is not affected by a DSoD constraint.

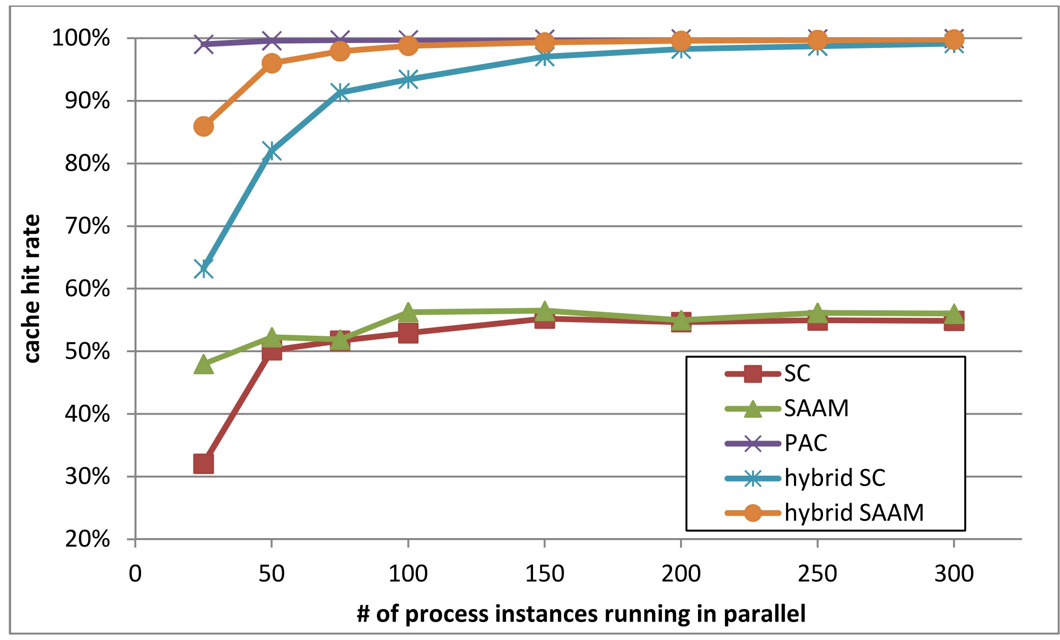


Figure 8-14: Dynamic access control (DSoD) - hit rate

PAC shows a similar behaviour as in the static scenario. This, however, comes not for free as the strategy requires that anticipated access requests during a process execution are pre-evaluated, independent whether these cached decisions will be used afterwards. We measured the additional effort required. Table 25 illustrates the results, i.e., showing the total time (in seconds) solely required for cache management by pre-evaluating anticipated access control requests. The time for cache management for PAC ranges from 119 s to 214 s, given the respective parallel execution of 25 to 300 process instances.

#process instances	PAC [s]	Hybrid SC [s]	Hybrid SAAM [s]
25	119	26	24
50	127	30	31
75	135	42	41
100	145	44	46
150	161	61	61
200	173	66	68
250	192	70	84
300	214	81	85

Table 25: Comparing accumulated cache management efforts for pre-evaluations of access control requests

The pre-evaluations of PAC include both static and dynamic access control requests which cause overheard. To reduce this overhead we introduce hybrid caching. Hybrid caching combines static caching strategies with the proactive strategy (cf. Section 7.2). We

distinguish between Hybrid SC (combining SC and PAC) and Hybrid SAAM (combining SAAM and PAC). The idea in both cases is that for static access control requests the static caching strategies are used (which do not require additional effort for cache management); for the dynamic access requests the cache and cache-update strategies of PAC are used.

Figure 8-13 shows that the static parts of hybrid caches still require a certain time until access requests can directly be answered from the cache. The cache hit rate, however, converges against 100% with increasing number of process instances (see Figure 8-14). This is fact for both cases, combining SC and PAC or SAAM and PAC. Also, it can be seen that given any number of process instances, the hybrid versions are always faster than their static counterparts. Table 25 shows that we achieved our main goal for hybrid caching: the reduction of the time required for cache management. The accumulated times for cache management for the hybrid caching strategies is just required to update its proactive cache in case there are access control requests which need the evaluation of dynamic context constraints.

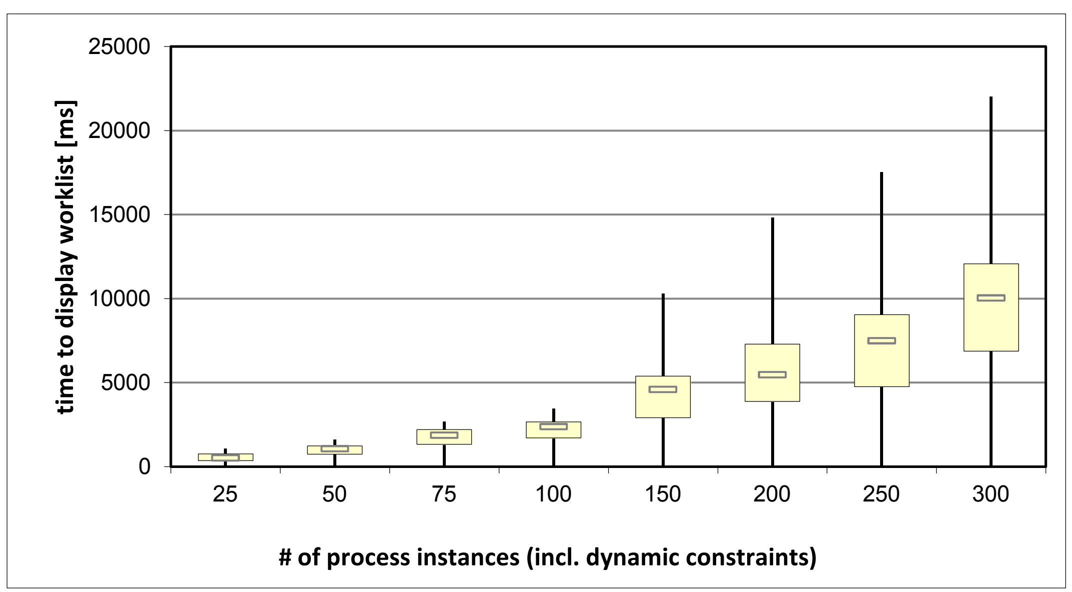


Figure 8-15: Dynamic access control - GWL No Cache

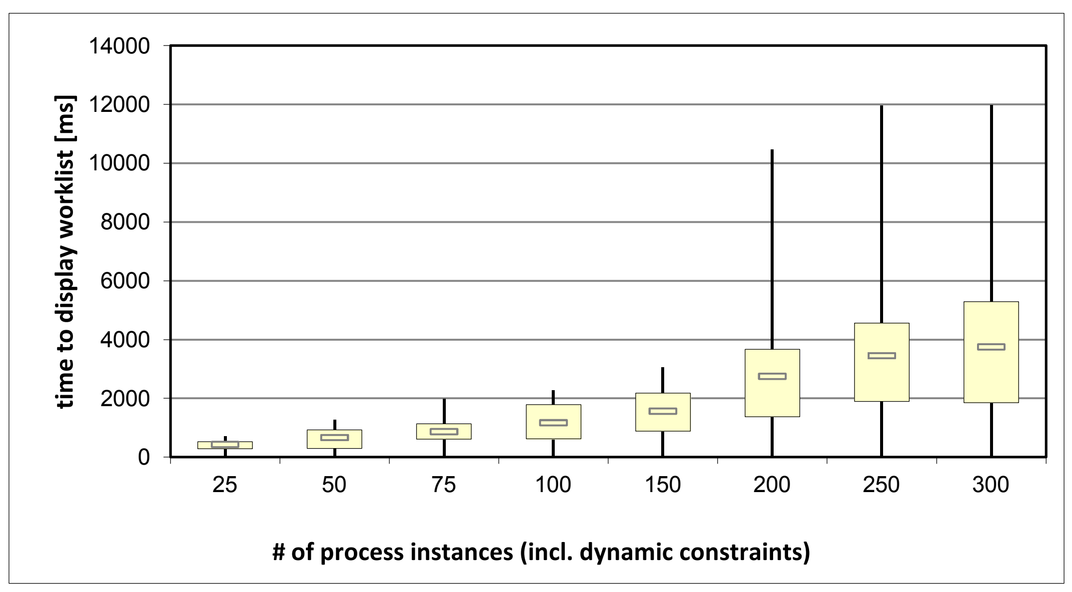


Figure 8-16: Dynamic access control - GWL SC

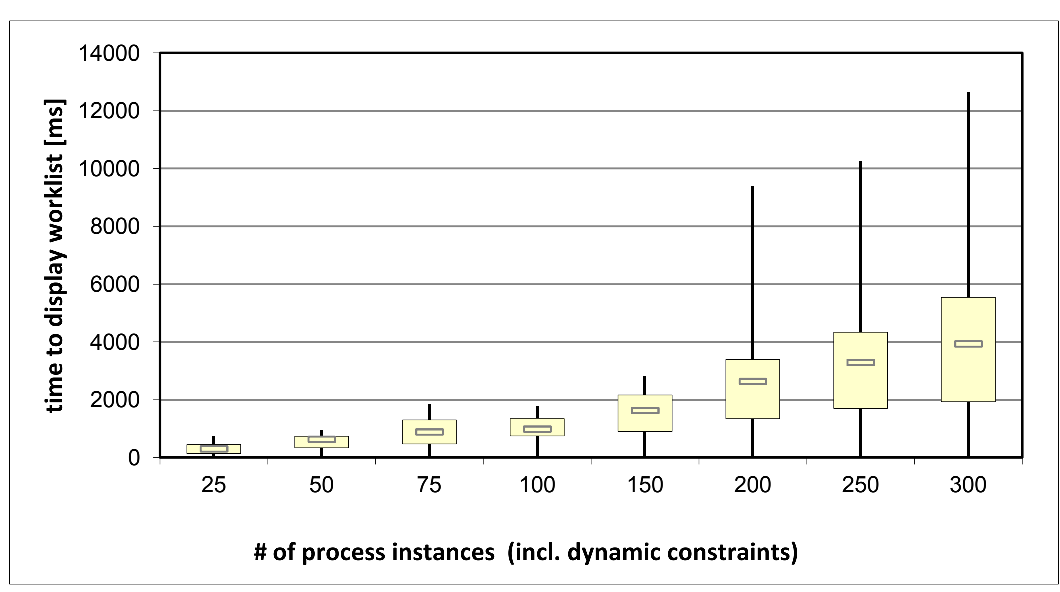


Figure 8-17: Dynamic access control - GWL SAAM

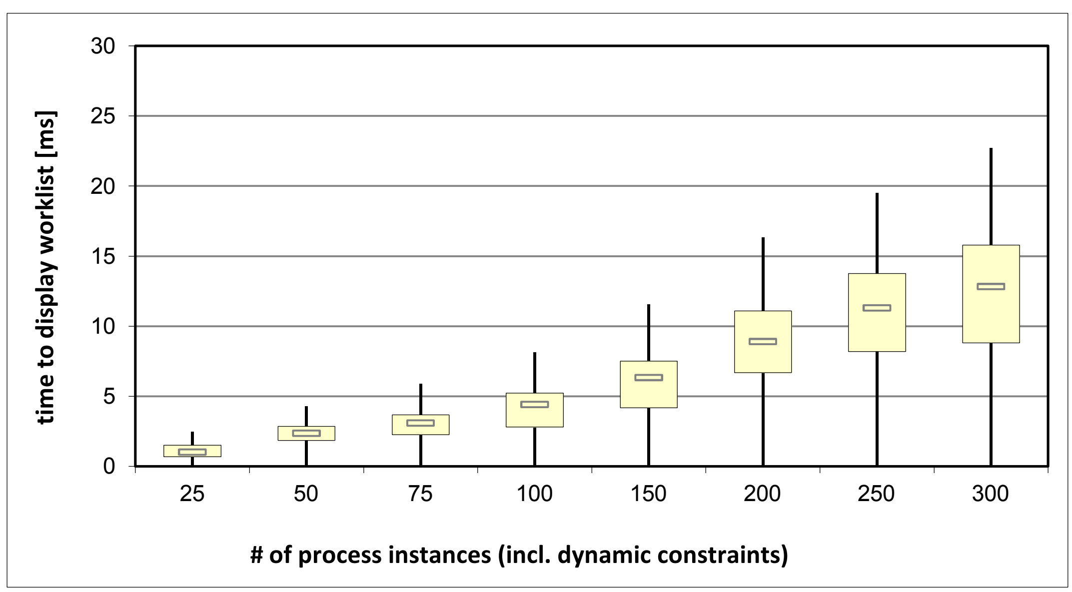


Figure 8-18: Dynamic access control - GWL PAC

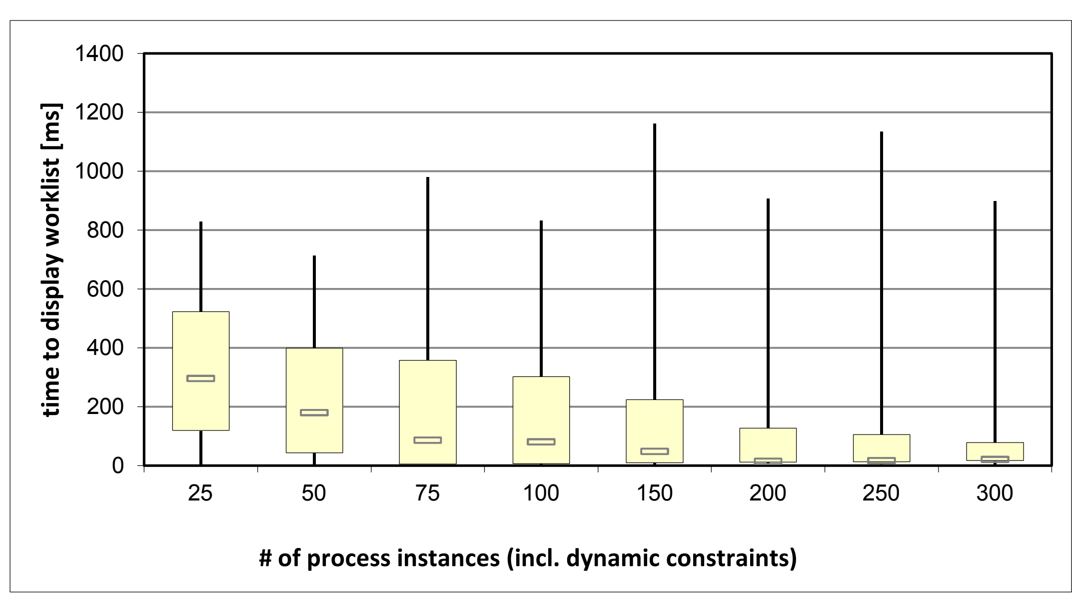


Figure 8-19: Dynamic access control - GWL Hybrid SC

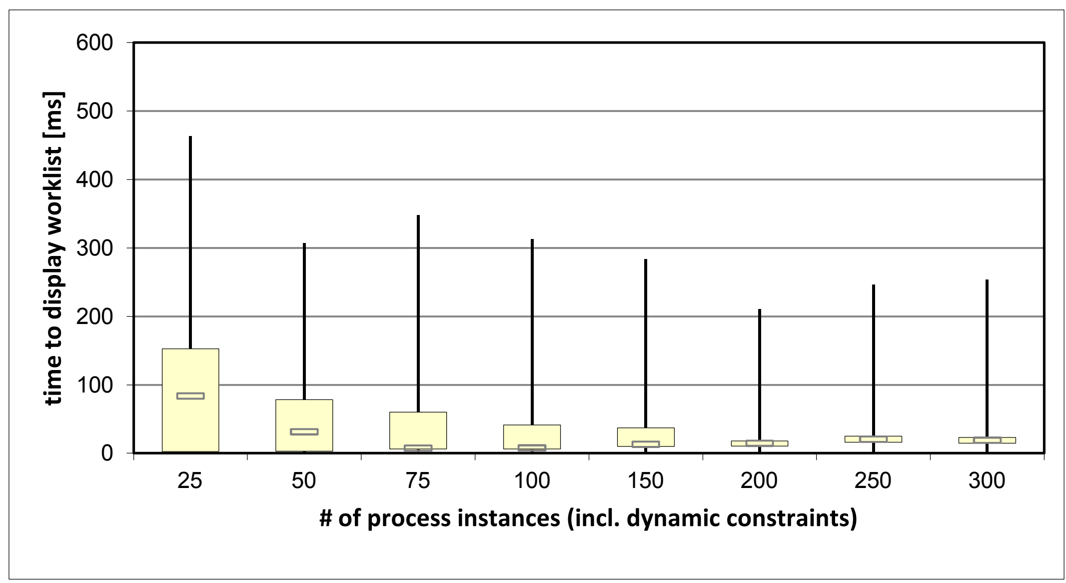


Figure 8-20: Dynamic access control - GWL Hybrid SAAM

The box-plots of Figure 8-15 to Figure 8-20 illustrate the different times it takes to display the worklist with caching and, as reference, without caching (see Figure 8-15). The latter shows that displaying the worklist takes more time if compared to static environments. This is the case as the evaluation of dynamic context constraints, i.e., DSoD constraints generally require the additional effort to fetch and evaluate dynamic context information, leading to a longer request evaluation time—compared to requests evaluated based on standard role-based policies.

The box-plots for the SC (see Figure 8-16) and SAAM (see Figure 8-17) show that the time to display the worklist increases with the number of instances in a system, rather than decreasing as it is the case in the static environment. This behaviour, again, results from the fact, that dynamic access control requests are always regularly evaluated. Given 200 and more process instances the difference between SC and SAAM is very small; the times required for the regular evaluations of dynamic access control requests clearly dominate the results.

PAC remains the fastest strategy within the complete tested scenario. The worklists can be displayed after a maximum of 23 ms. The two hybrid caches behave similar compared to SC and SAAM in the static environments (compare Figure 8-16 with Figure 8-8 and Figure 8-17 with Figure 8-9). The box-plots show, however, that the static parts of the caches require time to fill the cache. This warm-up period leads to possible times above 1 s in case of Hybrid SC and above 300 ms in case of Hybrid SAAM. In both cases, however, those spikes are quite rare as in 90% of all cases the time to display the worklist lies below 230 ms and 50 ms respectively.

Small and Large Process Definitions

For dynamic access control decisions we also conducted our analysis of business process executions for different sizes of process definitions. Below we depict the results for small and large process definitions. It can easily be seen, that the average response times are similar to the ones we reported on medium sized process definitions before.

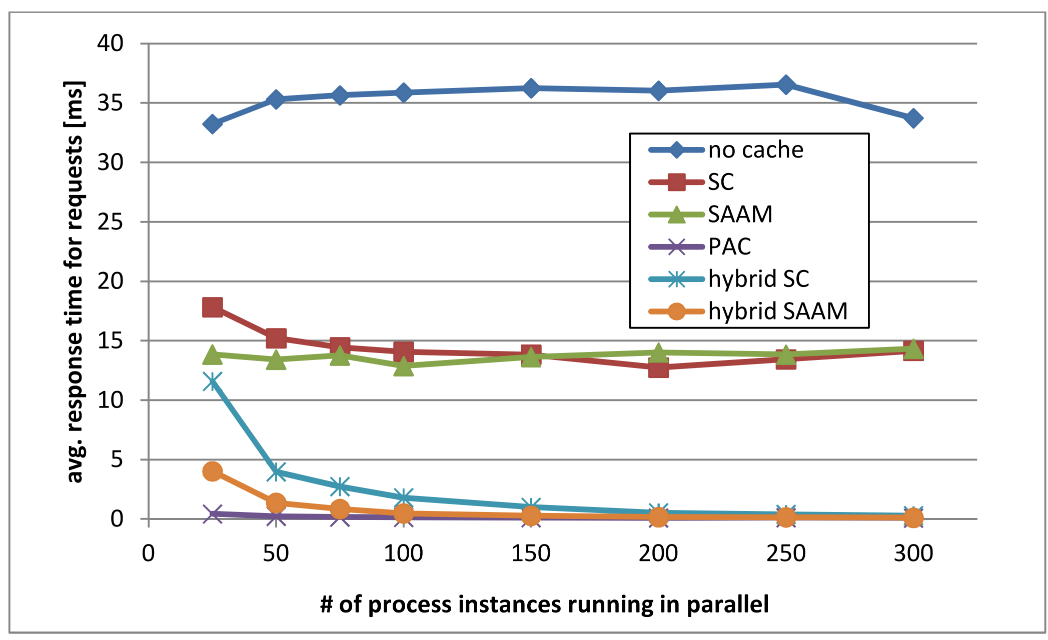


Figure 8-21: Dynamic access control (DSoD) - average access time for small business process definitions

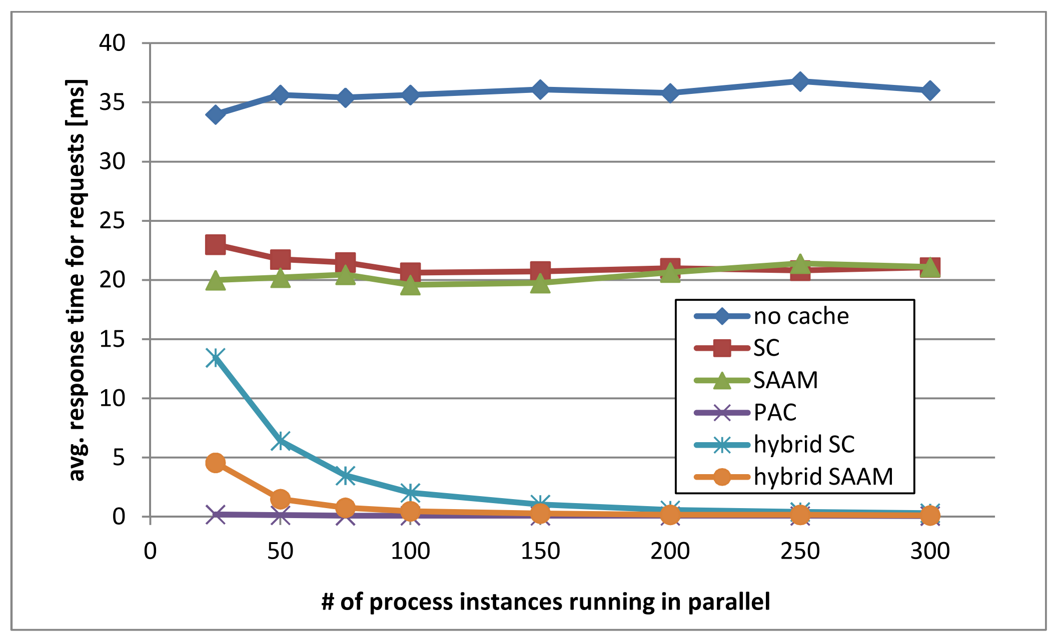


Figure 8-22: Dynamic access control (DSoD) - average access time for large business process definitions

Discussion

Overall, our experiments show that in dynamic environments the use of process and workflow models can increases the overall system performance. In particular, caching approaches that exploit these models perform significantly better than workflow and process model independent approaches such as SC or SAAM. In more detail, already a considerably low amount of dynamic context constraints (in our scenarios only 40% of the total resources were affected) result in a significant impact on response times. Dynamic access control requests cannot be cached and, hence, lead to increased cache misses. We expect that in future, especially for business process-driven environments, the general amount of dynamic context constraints will rather increase than decrease. An increasing amount of dynamic context constraints would lead to further increasing response times if no caching or static caching strategies are used.

In contrast, PAC provides that the response times remain very low, given the system allows the detection of context changes which can be used to update the cache. In our experience, the additional effort PAC requires for its cache management sums up to, e.g., 214 s for 300 process instances (cf. Table 25).

Of special interest are the results for our two-level caching architectures, combining both static and dynamic caching strategies. In dynamic environments, hybrid caching enables using the PAC update strategy for dynamic access control decisions as first level cache and static caching strategies as second level cache. Also in this case, however, the combined caches have a warm-up phase leading to situations where the time to display the worklist is up to a factor of 10 greater than the average time of the same scenario.

Based on the results for dynamic environments we can conclude that for systems requiring a very low response time, again PAC should be preferred and the additional effort required remains within a reasonable range. In all other cases a mixture between established caching strategies for static environments combined with the update strategy of PAC seems the preferred choice. Again, if a combination with the tested SAAM strategy is considered, the systems static policies should be based on RBAC.

8.5 Summary

In this chapter we presented our performance analysis of ProActive Caching (PAC). We selected different caching strategies, namely, standard caching, $SAAM_{RBAC}$ [68], as well as different hybrid approaches, to be compared against PAC. While standard caching represented a generic caching strategy, both, PAC and $SAAM_{RBAC}$ are specifically developed for caching access control decisions.

We presented the results of our analysis for both, static and dynamic environments. For each caching strategy and each environment, we showed the average response times required for access control requests, as well as the times required to query all access control requests to display the general worklist (GWL), given 25 to 300 process instances running in parallel.

Moreover, for each environment, we thoroughly discussed the outcome of our tests and gave recommendations which caching strategy is most suited for which practical system requirements or properties.

9 Conclusion & Future Research

In this chapter we discuss the results of our work and identify lines for future research. In Sections 9.1 we revise our goals and summarize our main results. In Section 9.3 we discuss our work compared to related work and present future lines of research in Section 9.3.

9.1 Conclusion

The research objective of this work was to provide a caching framework for business process-driven environments. In particular, the goal of this work was twofold:

- Our first objective was to significantly reduce the time required until access control decisions can be consumed and enforced by the respective BPMS.
- Our second objective was to provide a cache management strategy that allows to cache access control decisions which are based on constraints relying on dynamic context information.

For achieving our first goal we introduced ProActive Caching (PAC). PAC is able to provide very low access control response times. This is achieved by means of pre-fetching and caching those access control decisions, anticipated to be required during the execution of a business process. PAC allows that almost all first-time queries can already be answered by the cache, resulting in an almost 100% cache hit-rate, while only having those access control decisions in the cache related to currently active process instances.

For achieving the second goal we either provide means to constantly update cache entries if they become invalid due to changes of the context they rely on, or we enable the deferral of the evaluation of dynamic context constraints until the point the cached decision is actually requested. The evaluation of dynamic context constraints always requires fetching additional context information from different context providers, putting additional effort and time consumption into each access control evaluation. Updating still requires the effort for evaluating an access control request, the advantage is that at the point the cached decision is requested, an instant answer is delivered.

In the following we summarize the technical results of our work.

ProActive Caching

We presented *ProActive Caching* (PAC) in Chapter 5, a caching strategy which is specifically developed to the dynamic properties of business process-driven environments. PAC anticipates which access control requests will be queried during the execution of business processes based on underlying process models and pre-evaluates them accordingly.

We distinguish between the two categories of change-detectable context constraints and open constraints. This distinction allows either to update cached decisions upon a detected context change, or to postpone the evaluation of open constraints until a cached decision is actually consumed from the cache. Hence, our caching strategy enables the cache management to instantly react on detectable context changes; or, if context changes cannot be detected, we provide cache-internal mechanisms, allowing the consideration of dynamic context information while actually querying a decision from the cache.

Access control decisions based on dynamic context constraints usually require to be evaluated for a specific process instance. There are, however, access control decisions which are valid across different instances of a process. In Chapter 7 we presented an optimized version of PAC, showing how to enable caching of access control decisions across different process instances to reduce the amount and effort of pre-evaluations.

Hybrid caching strategy

We presented a novel, two-levelled caching strategy which allows the combination of our PAC strategy with other caching strategies; introduced in Chapter 7. Our hybrid approach uses PAC as first level cache, responsible for caching all access control decisions which require the evaluation of dynamic context constraints. All access control requests which do not fall under the responsibility of PAC are transparently forwarded to a second level cache which may implement any access control strategy which is capable of caching access control decisions.

Our two-levelled cache approach allows the combination of strategies such that drawbacks from one strategy can be compensated by the strengths of another one, and vice versa. Our experimental analysis supports this in Chapter 8.

Automatic Generation of Caching Heuristic

We presented means to automatically generate the caching heuristic required to generically define which access control requests should be pre-evaluated. This allows us to keep the administrational overhead for our caching approach very low. In fact, once PAC is implemented and initially configured with the system's life cycle events, business processes, business object and task relations, as well as the security policy, it is feasible to automatically generate the dependency relations required for our cache management, given the algorithms presented in Chapter 6.

Analysis and Comparison

We conducted a thorough analysis of our caching strategy according to performance improvements and provided a detailed comparison with other approaches for caching role-based and dynamic access control decisions in business process-driven systems

(cf. Chapter 8). In particular, we compared standard, generic caching against caching strategies developed to specifically cache access control decisions, i.e., $SAAM_{RBAC}$ and ProActive Caching. $SAAM_{RBAC}$ [68] is a caching strategy developed for systems which rely on role-based access control (RBAC) [50]. Moreover, we presented performance results on the combination of ProActive Caching with other caching strategies, using our two-levelled caching approach.

Generic Caching Architecture

Access control architectures are typically implemented using the request-response paradigm [27, 44, 46], where a policy enforcement point (PEP) sends access control requests to an access control decision point (PDP), and enforces its answers respectively. We presented an enriched general architecture (see Chapter 5) such that ProActive Caching is embedded between a PEP and PDP, allowing to transparently returning cached access control decisions to the PEP. On the other side, the cache management makes use of the link to the PDP to pre-evaluate access control decisions to be stored in the cache. Moreover, our architecture does not depend on the implemented caching strategy: generic caching strategies (e. g., [36]) can be as easily integrated as access control specific ones (e. g., [31, 68]).

9.2 Discussion on Updating Cache Entries

There is a large body of literature concerned with improving the performance of evaluating access control requests in IT systems (e.g., by optimizing the evaluation engine or restructuring access control policies [34, 35, 39]), and using caching approaches in particular [6, 8, 15, 29-31, 68, 69]. To our knowledge, however, there is none which specifically developed caching strategies to allow caching and updating access control decisions which have been based on the evaluation of dynamic context constraints, such as the ones we introduced in Chapter 3 (i.e., Dynamic Separation of Duties, Binding of Duties, etc.).

One major challenge is dealing with access control decisions possibly becoming invalid over time if they rely on dynamic context. Our approach is to either update cache entries upon pre-defined events or storing OCs into the cache entry for evaluation upon the consumption of the cached decision.

A situation which further requires cache entries to be updated is a change of the security policy. Our analysis assumed that the security policy remains unchanged. This includes no changes on role-user or role-permission assignments. In general, however, our tested caching strategies could handle policy changes in different ways. One possibility for all caching strategies is to flush the cache upon a policy change. Already cached decisions are vanished and have to be re-evaluated. Especially for the standard caching strategy, each flush would result in a warm-up phase with a low hit-rate until the cache slowly regains content.

For ProActive Caching, this would result in cache misses for those access control requests for which access control decisions have already been pre-computed and stored in the cache, hence, for all tasks which are currently active or which are part of the "fringe" (i.e., all immediate upcoming tasks, cf. Section 5.3.1). New cache entries are created with every broadcasted event defined in dependency relations such that we would only expect a small number of cache entries not being available.

Other possibilities are specific to a respective caching strategy. The caching strategy $SAAM_{RBAC}$ [68], for instance, explicitly handles updates of role-based security policies by sending update massages to the caching component whenever a role definition changes. The update message is constructed in such a way that the same algorithms are used which usually process and incorporate regular access control decisions into the cache.

Also for PAC there is another way of handling changes in the security policy than just flushing the cache: PAC may use its dependency relations to update the cache upon policy updates. PAC reacts on broadcasted events which - in our work - are broadcasted by the BPMS. Other components, however, may broadcast events as well, such as the component responsible for managing the security policy. Dependency relations - and revocation targets in particular - may be defined in such a way that specific cache entries are revoked or updated upon events indicating that a specific part of the security policy has changed.

9.3 Future Research

We see several lines of future research:

While we only tested scenarios for caching access control decisions, we are convinced that our caching strategy may also hold for caching other types of information required during the execution of business processes. Examples are internal or external context information about the availability of users or other resources, or specific data objects from back-end systems that are required for performing a specific task. Instead of access control decisions, these types of information could be pre-fetched such that they are already available if the BPMS requires them.

The influence of different dynamic aspects of modern business applications needs to be investigated. Modern enterprises have different business process models actively in use. Some of them are regularly executed (e.g., purchase orders), others are only executed once every month (e.g., payroll processes). Such additional dynamic behavior could be considered to specifically optimize pre-fetching of access control decisions accordingly.

For the dynamic access control requirements, a fine-grained classification of constraint types requiring dynamic context information may increase the overall efficiency of caches. This requires a throughout study of high-level business requirements like Basel II [2] for

identifying the different dynamic aspects required, followed by their formalization and integration into existing policy frameworks.

Finally, the most efficient caching strategies in our analysis (PAC and SAAM) depend on the underlying access control model. Thus, it seems to be interesting to extend our comparison to different access control models besides RBAC. Here, multi-level security models such as Bell-LaPadula [3] seem to be a particularly interesting target. While for SAAM there exists already a variant supporting Bell-LaPadula (see [15] for details), our caching heuristics (i.e., dependency relations) would need to be adapted.

10 References

[1] ANSI INCITS 259-2004 for role based access control, 2004.

[2] Basel Committee on Banking Supervision. Basel II: International convergence of capital measurement and capital standards. http://www.bis.org/publ/bcbsca.htm, Basel, Switzerland, 2004.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model, volume II. In *Journal of Computer Security 4*, pages 229–263, 1996. An electronic reconstruction of *Secure Computer Systems: Mathematical Foundations*, 1973.

[4] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

[5] E. Bertino, E. Ferrari, and V. Atluri. The specification and enforcement of authorization constraints in workflow management systems. *ACM Transactions on Information and System Security*, 2:65–104, February 1999.

[6] K. Beznosov. Flooding and recycling authorizations. In *NSPW '05: Proceedings of the 2005 workshop on New security paradigms*, pages 67–72, New York, NY USA, 2005. ACM Press.

[7] BonitaSoft. Bonita Open Solution. http://www.bonitasoft.com/, 2009.

[8] K. Borders, X. Zhao, and A. Prakash. CPOL: high-performance policy evaluation. In *ACM Conference on Computer and Communications Security*, pages 147–157, 2005.

[9] R. A. Botha and J. H. P. Eloff. Designing role hierarchies for access control in workflow systems. In *COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, pages 117–122, Washington, DC, USA, 2001. IEEE Computer Society.

[10] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual modeling of workflows. *OOER '95: Object-Oriented and Entity-Relationship Modeling*, pages 341–354, 1995.

[11] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *International conference on Very Large Data Bases (VLDB)*, pages 127–141. VLDB Endowment, 1985.

[12] A. Cichocki, M. Rusinkiewicz, and D. Woelk. *Workflow and Process Automation: Concepts and Technology*. Kluwer Academic Publishers, Norwell, MA, USA, 1998.

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

[14] J. Crampton and H. Khambhammettu. Delegation and satisfiability in workflow systems. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 31–40, New York, NY USA, 2008. ACM Press.

[15] J. Crampton, W. Leung, and K. Beznosov. The secondary and approximate authorization model and its application to bell-lapadula policies. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 111–120, New York, NY USA, 2006. ACM Press.

[16] T. Curran, G. Keller, and A. Ladd. *SAP R/3 business blueprint: understanding the business process reference model*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[17] D. Duchamp. Prefetching hyperlinks. In *USITS'99: Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems*, pages 12–12, Berkeley, CA, USA, 1999. USENIX Association.

[18] R. K. D. Ferraiolo. Role-based access control. 1992.

[19] Forrester Research. The EA view: BPM has become mainstream, 2008.

[20] Forrester Research. The Forrester Wave: Integration-centric business process management suites, Q4 2008, 2008.

[21] C. Fox and P. Zonneveld. *it Control Objectives for Sarbanes-Oxley: The Role of it in the Design and Implementation of Internal Control Over Financial Reporting*. IT Governance Institute, Rolling Meadows, IL, USA, 2nd edition, Sept. 2006.

[22] R. Goodwin, S. F. Goh, and F. Y. Wu. Instance-level access control for business-to-business electronic commerce. *IBM Syst. J.*, 41(2):303–317, 2002.

[23] Google. Web cache. http://www.google.com/intl/en/help/features_list.html#cached, 2010.

[24] J. A. Hoxmeier and C. DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. In *AMCIS Americas Conference of Information Systems*, pages 10–13, 2000.

[25] W.-K. Huang and V. Atluri. SecureFlow: a secure web-enabled workflow management system. In *RBAC '99: Proceedings of the fourth ACM workshop on Role-based access control*, pages 83–94, New York, NY, USA, 1999. ACM.

[26] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

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

[28] K. Knorr and H. Stormer. Modeling and analyzing separation of duties in workflow environments. In *Sec '01: Proceedings of the 16th international conference on Information security: Trusted information*, pages 199–212, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[29] M. Kohler, A. D. Brucker, and A. Schaad. Proactive caching: Generating caching heuristics for business process environments. In *International Conference on Computational Science and Engineering (CSE)*, pages 207–304. IEEE Computer Society, Washington, DC, USA, Aug. 2009.

[30] M. Kohler and R. Fies. ProActive Caching - a framework for performance optimized access control evaluations. In *POLICY '09: Proceedings of the 2009 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 92–94, Washington, DC, USA, 2009. IEEE Computer Society.

[31] M. Kohler and A. Schaad. Proactive access control for business process-driven environments. In *Proceedings of the 2008 Annual Computer Security Applications Conference, ACSAC 2008*, pages 153–162, Washington, DC, USA, 2008. IEEE Computer Society.

[32] H. Koshutanski and F. Massacci. An access control framework for business processes for web services. In *XMLSEC '03: Proceedings of the 2003 ACM workshop on XML security*, pages 15–24, New York, NY USA, 2003. ACM Press.

[33] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*, pages 195–201. ACM Press, 1998.

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

[35] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran. Statistics & clustering based framework for efficient XACML policy evaluation. In *IEEE International Workshop on Policies for Distributed Systems and Networks*, volume 0, pages 118–125, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[36] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association.

[37] J. Mendling. *Detection and Prediction of Errors in EPC Business Process Models*. PhD thesis, Vienna University of Economics and Business Administration (WU Wien), Austria, 2007.

[38] R. B. Miller. Response time in man-computer conversational transactions. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, New York, NY, USA, 1968. ACM.

[39] P. Miseldine. Automated XACML policy reconfiguration for evaluation optimisation. In *SESS*, pages 1–8, 2008.

[40] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 62–73. ACM Press, 1992.

[41] J. Mueller. *Workflow-based Integration*. Xpert.press. Springer-Verlag, Heidelberg, 2005.

[42] J. Nielsen. *Usability Engineering*. Academic Press, Boston, Mass., 1993.

[43] OASIS. Organization for the advancement of structured information standards. http://www.oasis-open.org, 1993.

[44] OASIS. eXtensible Access Control Markup Language 2.0 (XACML). www.oasis-open.org/committees/xacml, 2005.

[45] O. M. G. OMG. BPMN: Business process model and notation specification. Specification document, Object Management Group, 2009.

[46] ORKA. Organisatorische Kontrollarchitektur. http://www.organisatorische-kontrolle.de, 2006.

[47] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, New York, NY, USA, 1995. ACM.

[48] S. Sackmann and J. Strüker. *Electronic Commerce Enquête 2005: 10 Jahre Electronic Commerce - Eine stille Revolution in deutschen Unternehmen*. Konradin IT-Verlag, 2005.

[49] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[50] 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, New York, NY USA, 2000. ACM Press.

[51] SAP AG. SAP NetWeaver BPM. White paper, SAP AG, January 2009.

[52] SAP AG. http://www.sap.com, 2010.

[53] P. Sarbanes, G. Oxley, et al. Sarbanes-Oxley Act of 2002, 2002.

[54] A. Schaad, V. Lotz, and K. Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 139–149, New York, NY USA, 2006. ACM Press.

[55] A. Schaad, P. Spadone, and H. Weichsel. A case study of separation of duty properties in the context of the austrian "eLaw" process. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1328–1332, New York, NY USA, 2005. ACM Press.

[56] A. Scheer. *ARIS - vom Geschäftsprozess zum Anwendungssystem*. Springer-Verlag, Heidelberg, 2002.

[57] W. Shin and H. K. Kim. A simple implementation and performance evaluation extended-role based access control. In *International Conference on Software Engineering, Parallel & Distributed Systems (SEPADS)*, pages 1–5, Stevens Point, Wisconsin, USA, 2005. World Scientific and Engineering Academy and Society (WSEAS).

[58] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, Berkeley, CA, USA, 1999. USENIX Association.

[59] V. Stiehl and J. Deng. Project issue management. Software developer network (sdn) article, SAP AG, December 2008.

[60] SUN. XACML reference implementation. http://sunxacml.sourceforge.net/, 2006.

[61] K. Tan, J. Crampton, and C. A. Gunter. The consistency of task-based authorization constraints in workflow systems. In *CSFW '04: Proceedings of the 17th IEEE workshop*

on Computer Security Foundations, Washington, DC, USA, 2004. IEEE Computer Society.

[62] The JBoss Group. jBPM Version 3.2. http://www.jboss.com/products/jbpm, 2008.

[63] R. K. Thomas. Team-based access control (tmac): a primitive for applying role-based access controls in collaborative environments. In *RBAC '97: Proceedings of the second ACM workshop on Role-based access control*, pages 13–19, New York, NY USA, 1997. ACM Press.

[64] R. K. Thomas and R. S. Sandhu. Towards a task-based paradigm for flexible and adaptable access control in distributed applications. In *NSPW '92-93: Proceedings on the 1992-1993 workshop on New security paradigms*, pages 138–142, New York, NY USA, 1993. ACM Press.

[65] R. K. Thomas and R. S. Sandhu. Task-based authorization controls (tbac): A family of models for active and enterprise-oriented autorization management. In *Proceedings of the IFIP TC11 WG11.3 Eleventh International Conference on Database Securty XI*, pages 166–181, London, UK, UK, 1998. Chapman & Hall, Ltd.

[66] W. Tscheschner. Transformation from EPC to BPMN. Technical report, Hasso-Plattner-Institute (HPI), 2010.

[67] J. Warner and V. Atluri. Inter-instance authorization constraints for secure workflow management. In *SACMAT '06: Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 190–199, New York, NY USA, 2006. ACM Press.

[68] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in RBAC systems. In *SACMAT '08: ACM symposium on Access control models and technologies*, pages 63–72, New York, NY USA, 2008. ACM Press.

[69] Q. Wei, M. Ripeanu, and K. Beznosov. Cooperative secondary authorization recycling. In *HPDC '07: Proceedings of the 16th international symposium on High performance distributed computing*, pages 65–74, New York, NY USA, 2007. ACM Press.

[70] M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, Heidelberg, 2007.

[71] W. M. C. WfMC. The workflow reference model. Technical report, Workflow Management Coalition, 1995.

[72] W. M. C. WfMC. Terminology & Glossary. Technical Report WFMC-TC-1011, Issue 2.0, Workflow Management Coalition, June 1999.

11 Appendix - EPC to BPMN Transformation

For our performance analysis we used standard processes provided by the reference model of the SAP R/3 system (e.g., found in Curran et. al [16], or Mendling [37]). These processes are modelled as Event-driven Process Chains (EPCs). We used a straight forward approach for transforming them into BPMN notation [45], illustrated by Tscheschner [66].

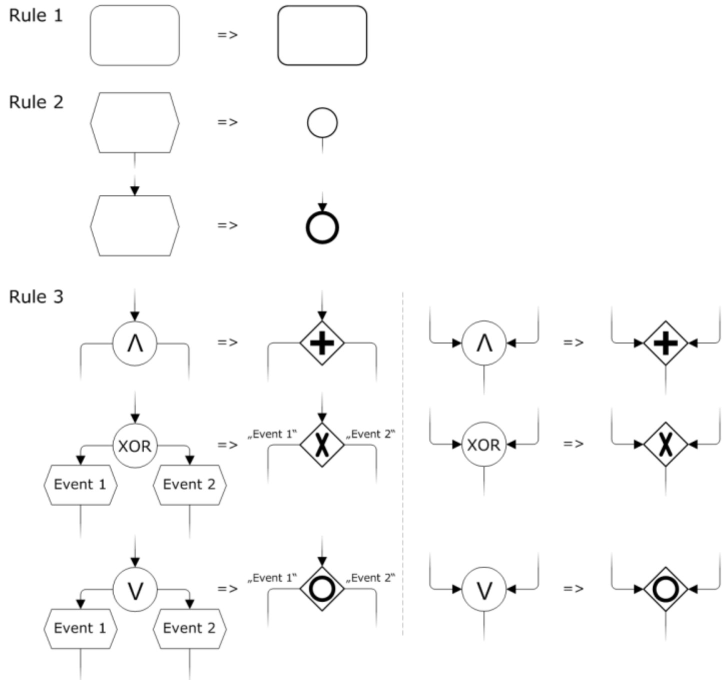


Figure 11-1: EPC to BPMN transformation rules [66]

Figure 11-1 depicts the respective transformation rules taken from [66]. On the left side, EPC elements are depicted, the right side shows the respective counterparts for BPMN. EPC functions (rectangles) are transformed into BPMN tasks, EPC events (hexagons) with outgoing flow are mapped to start events, events with incoming flow are mapped to end events, and all other events are neglected. Neglecting EPC events is feasible as they reflect status changes within a BPMS, influencing the further execution path of the process. The execution in BPMN is solely determined by its structural elements (i.e., AND-gates, OR-gates, XOR-gates); conditions which may in EPC be reflected by occurring events, includes BPMN into these structural elements.

In the following, we give an example of a process defined with EPC and its transformed version defined with BPMN (cf. Figure 11-2 and Figure 11-3). The EPC process is taken

from [37]. Moreover, we give further examples for processes we transformed and used for our analysis with Figure 11-6, Figure 11-4, and Figure 11-5.

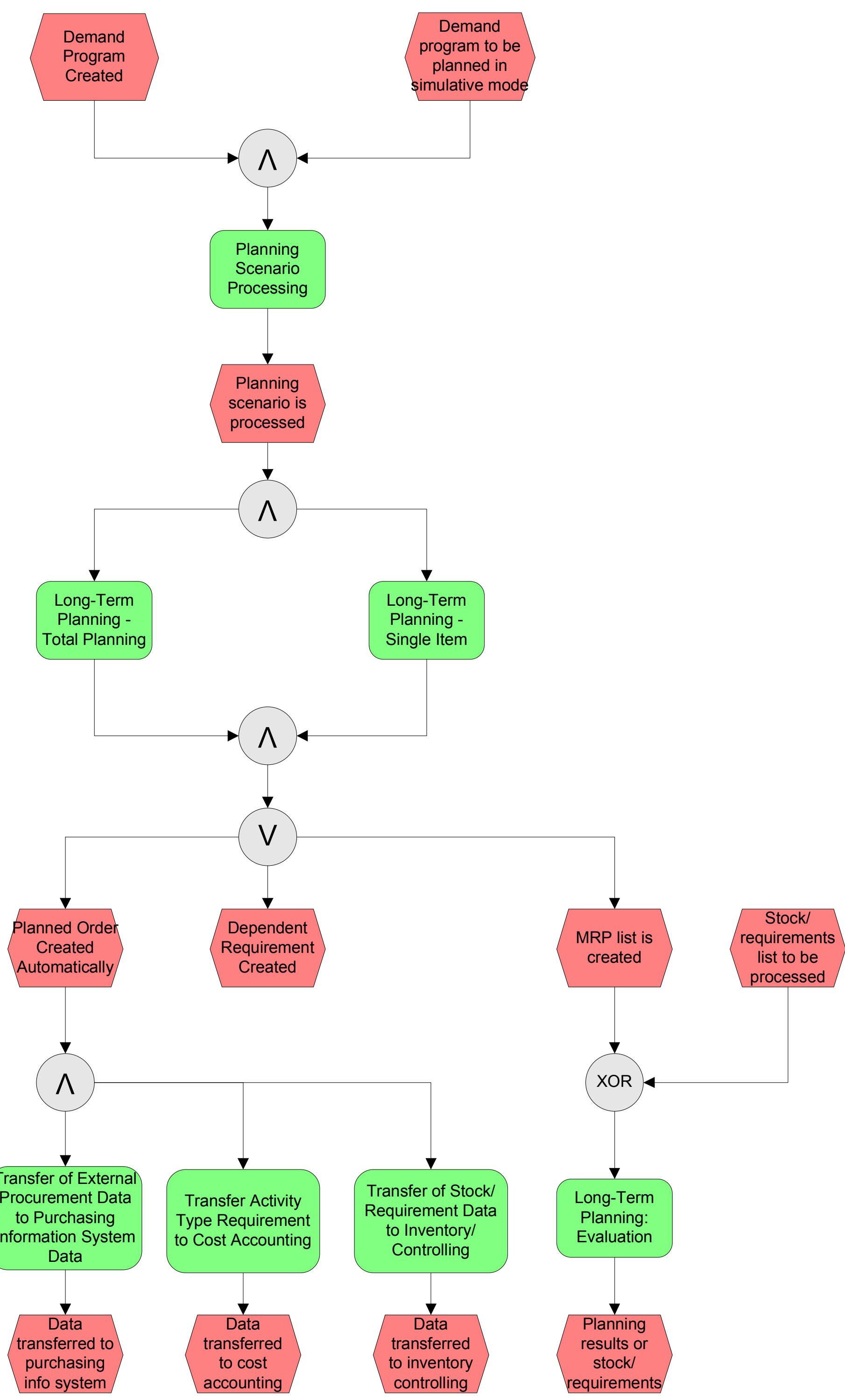


Figure 11-2: EPC: Production Planning and Procurement Planning - Market-Orientation Planning - Long-Term Planning (depicted in [37], page 348)

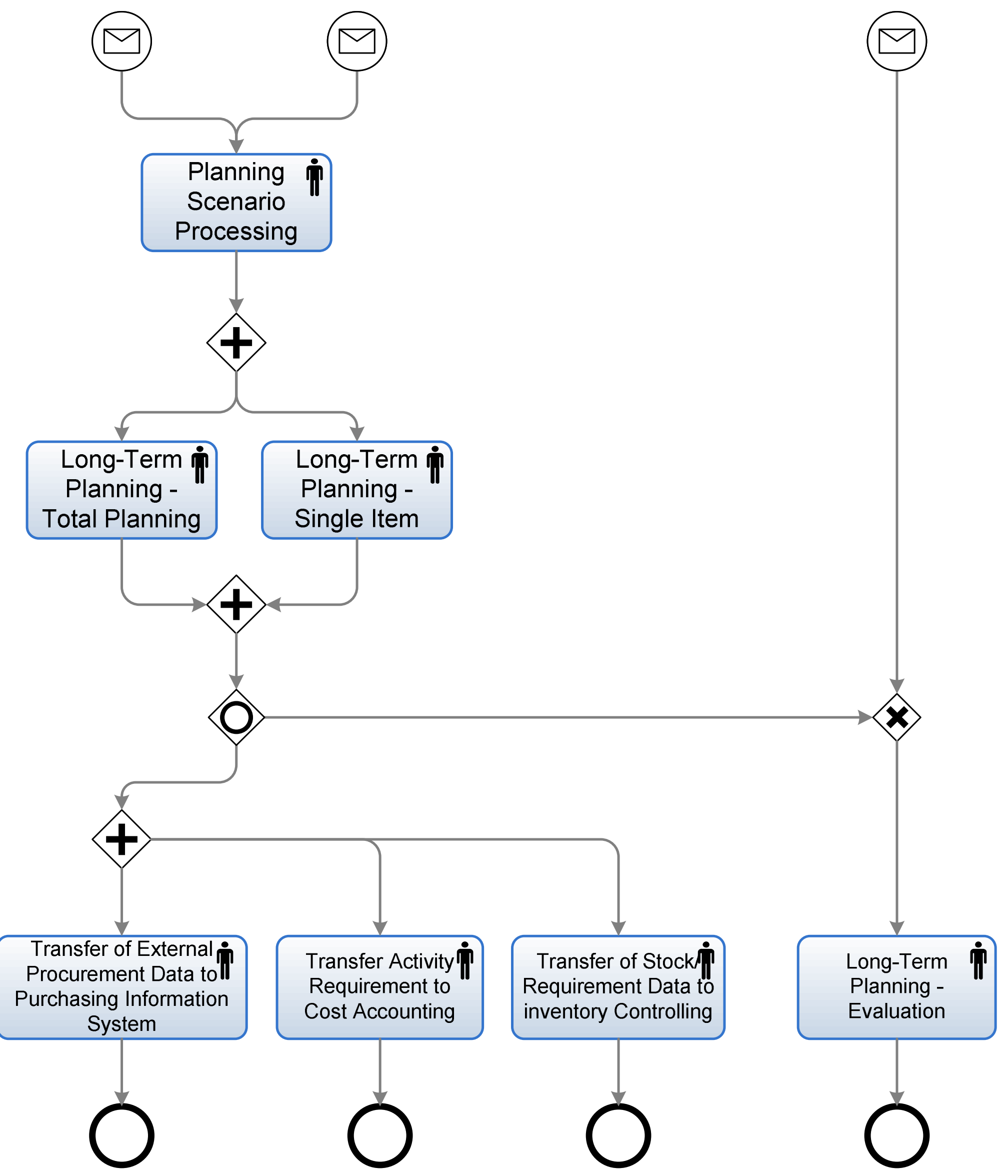


Figure 11-3: BPMN: Production Planning and Procurement Planning - Market-Orientation Planning - Long-Term Planning (7 tasks, original EPC process depicted in [37], page 348)

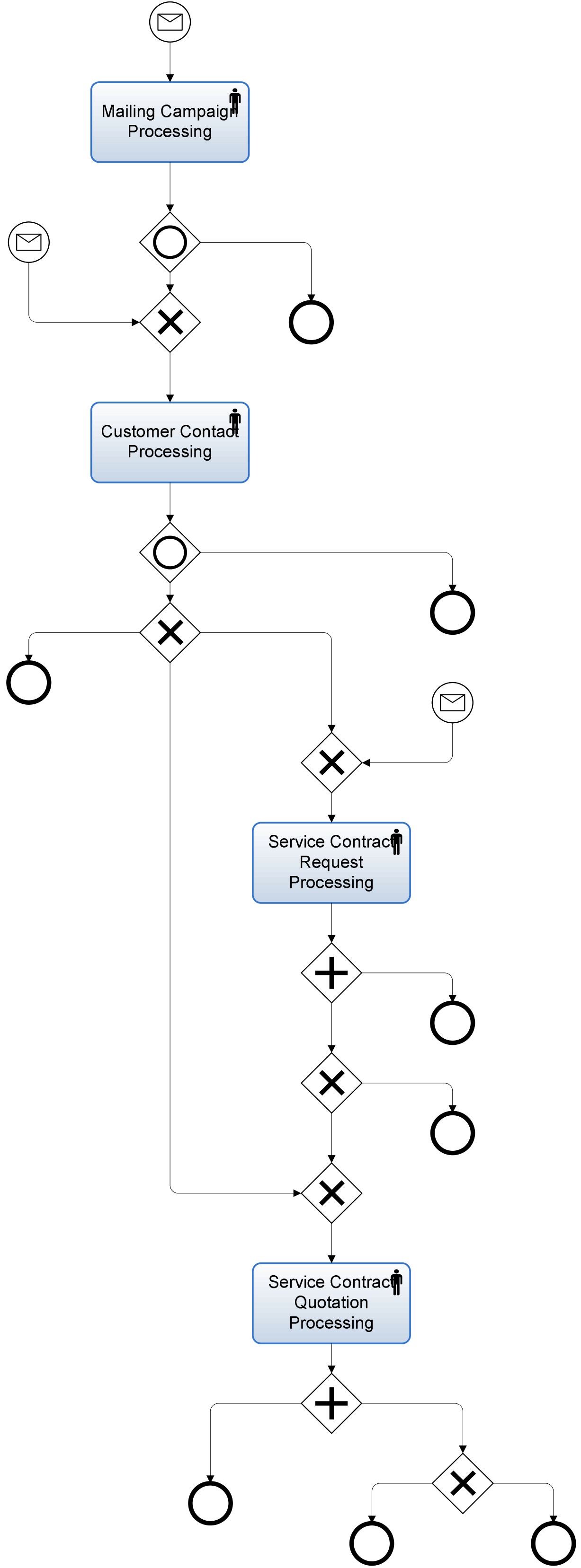


Figure 11-4: BPMN: Customer Service - Long-term Service Agreements - Presales Activities (4 tasks, original EPC process depicted in [37], page 322)

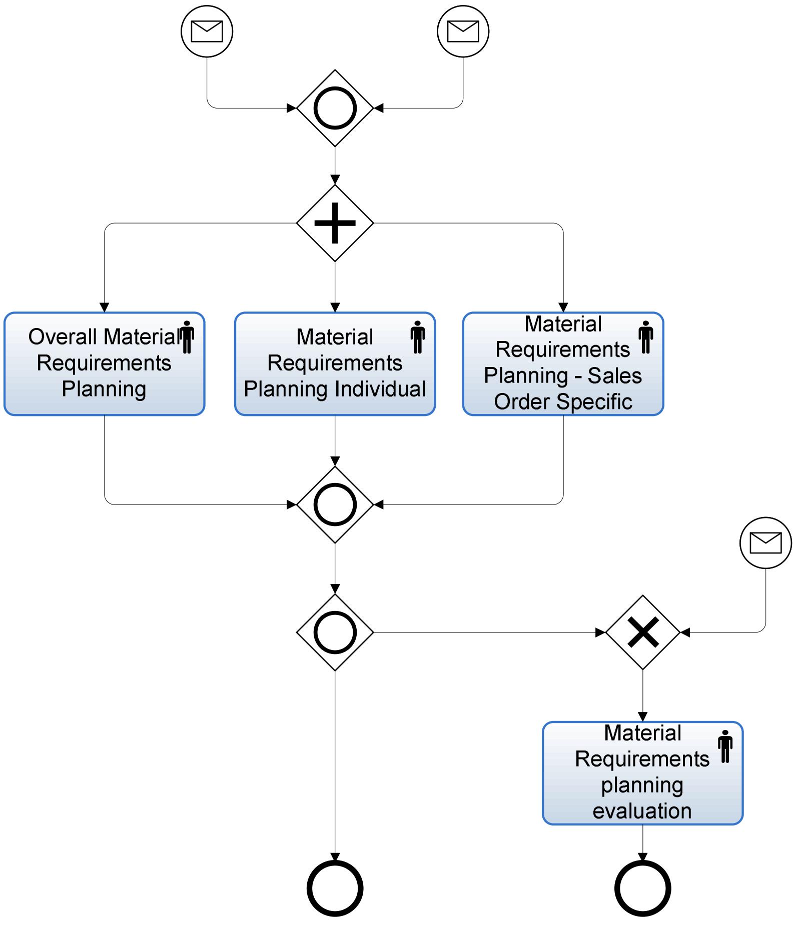


Figure 11-5: BPMN: Production Planning and Procurement Planning - Sales Order Oriented Planning - Material Requirements Planning (4 tasks, original EPC process depicted in [37], page 353)

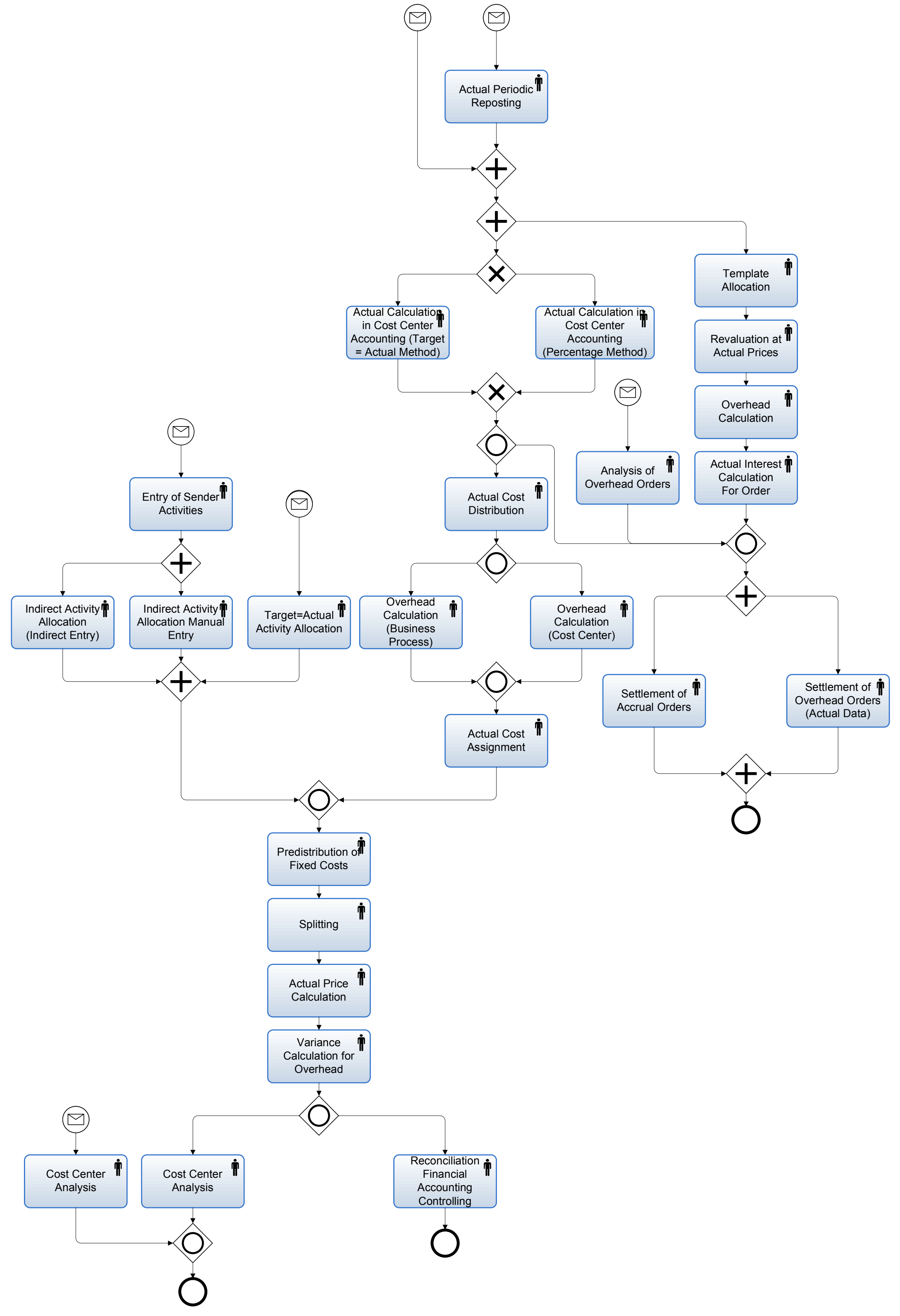


Figure 11-6: BPMN: Revenue and Cost Controlling - Period-End Closing (Controlling) - Period-End Closing Overhead Cost Controlling (25 tasks, original EPC process depicted in [37], page 366)