Anja Pölck

Small TCBs of Policy-controlled Operating Systems

Small TCBs of Policy-controlled Operating Systems

Anja Pölck

Universitätsverlag Ilmenau
2014

Impressum

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Angaben sind im Internet
über http://dnb.d-nb.de abrufbar.

Diese Arbeit hat der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau als Dissertation vorgelegen.

Tag der Einreichung:	1. Juli 2013
1. Gutachter:	Prof. Dr.-Ing. habil. Winfried E. Kühnhauser (Technische Universität Ilmenau)
2. Gutachter:	Prof. Dr. Elisa Bertino (Purdue University, West Lafayette, Indiana, USA)
3. Gutachter:	Prof. Dr.-Ing. habil Wolfgang Fengler (Technische Universität Ilmenau)
Tag der Verteidigung:	19. Dezember 2013

Technische Universität Ilmenau/Universitätsbibliothek
Universitätsverlag Ilmenau
Postfach 10 05 65
98684 Ilmenau
www.tu-ilmenau.de/universitaetsverlag

Herstellung und Auslieferung
Verlagshaus Monsenstein und Vannerdat OHG
Am Hawerkamp 31
48155 Münster
www.mv-verlag.de

ISBN 978-3-86360-090-7 (Druckausgabe)
URN urn:nbn:de:gbv:ilm1-2013000632

Titelphoto: photocase.com

Abstract

IT systems with advanced security requirements increasingly apply problem-specific security policies for describing, analyzing, and implementing security properties. Security policies are a vital part of a system's trusted computing base (TCB). Hence, both correctness and tamper-proofness of a TCB's implementation are essential for establishing, preserving, and guaranteeing a system's security properties.

Today's operating systems often show that implementing security policies is a challenge; for more than forty years, they have provided only a rather elementary support for discretionary, identity-based access control policies. As a consequence, major parts of the applications' security policies are implemented by the applications themselves, resulting in large, heterogeneous, and distributed TCB implementations. Thus, precisely identifying a TCB's functional perimeter is hard, which renders essential TCB properties – correctness, robustness, and tamper-proofness – difficult to achieve.

Efforts have been made to re-collect the policy components of operating systems and applications into a central component. So called policy-controlled operating systems provide kernel abstractions for security policies along with a policy decision and enforcement environment to protect and enforce the policies. Current policy-controlled operating systems are based on monolithic architectures so that their policy enforcement mechanisms are distributed all over the kernel. Additionally, they share the ambition to provide support for a wide variety of security policies that leads to universal policy decision and enforcement environments. Both results in large, complex, and expensive operating system TCBs, whose functional perimeter can hardly be precisely identified. As a consequence, a TCB's essential properties are hard to ensure in its implementation.

This dissertation follows a different approach based on the idea of methodically engineering TCBs by tailoring their policy decision and enforcement environment to support only those security policies that are actually present in a TCB. A TCB's functional perimeter is identified by exploiting causal dependencies between security policies and TCB functions, which results in causal TCBs that contain exactly those functions that are necessary to establish, enforce, and protect their policies. The precise identification of a TCB's functional perimeter allows for implementing a TCB in a safe environment that indeed can be isolated from untrusted system components. Thereby, causal TCB engineering sets the course for implementations whose size and complexity pave the way for analyzing and verifying a TCB's correctness and tamper-proofness. The application scenarios for causal TCB engineering range from embedded systems and policy-controlled operating systems to database management systems in large information systems.

Zusammenfassung

IT Systeme mit qualitativ hohen Sicherheitsanforderungen verwenden zur Beschreibung, Analyse und Implementierung ihrer Sicherheitseigenschaften zunehmend problemspezifische Sicherheitspolitiken, welche ein wesentlicher Bestandteil der Trusted Computing Base (TCB) eines IT Systems sind. Aus diesem Grund sind die Korrektheit und Unumgehbarkeit der Implementierung einer TCB entscheidend, um die geforderten Sicherheitseigenschaften eines Systems herzustellen, zu wahren und zu garantieren.

Viele der heutigen Betriebssysteme zeigen, welche Herausforderung die Realisierung von Sicherheitspolitiken darstellt; seit mehr als 40 Jahren unterstützen sie wahlfreie identitätsbasierte Zugriffssteuerungspolitiken nur rudimentär. Dies führt dazu, dass große Teile der Sicherheitspolitiken von Anwendersoftware durch die Anwendungen selbst implementiert werden. Infolge dessen sind die TCBs heutiger Betriebssysteme groß, heterogen und verteilt, so dass die exakte Bestimmung ihres Funktionsumfangs sehr aufwendig ist. Im Ergebnis sind die wesentlichen Eigenschaften von TCBs – Korrektheit, Robustheit und Unumgehbarkeit – nur schwer erreichbar.

Dies hat zur Entwicklung von Politik gesteuerten Betriebssystemen geführt, die alle Sicherheitspolitiken eines Betriebssystems und seiner Anwendungen zentral zusammenfassen, indem sie Kernabstraktionen für Sicherheitspolitiken und Politiklaufzeitumgebungen anbieten. Aktuelle Politik gesteuerte Betriebssysteme basieren auf monolithischen Architekturen, was dazu führt, dass ihre Komponenten zur Durchsetzung ihrer Politiken im Betriebssystemkern verteilt sind. Weiterhin verfolgen sie das Ziel, ein möglichst breites Spektrum an Sicherheitspolitiken zu unterstützen. Dies hat zur Folge, dass ihre Laufzeitkomponenten für Politikentscheidung und -durchsetzung universal sind. Im Ergebnis sind ihre TCB-Implementierungen groß und komplex, so dass der TCB-Funktionsumfang nur schwer identifiziert werden kann und wesentliche Eigenschaften von TCBs nur mit erhöhtem Aufwand erreichbar sind.

Diese Dissertation verfolgt einen Ansatz, der die TCBs Politik gesteuerter Betriebssysteme systematisch entwickelt. Die Idee ist, das Laufzeitsystem für Sicherheitspolitiken so maßzuschneidern, dass nur die Politiken unterstützt werden, die tatsächlich in einer TCB vorhanden sind. Dabei wird der Funktionsumfang einer TCB durch kausale Abhängigkeiten zwischen Sicherheitspolitiken und TCB-Funktionen bestimmt. Das Ergebnis sind kausale TCBs, die nur diejenigen Funktionen enthalten, die zum Durchsetzen und zum Schutz der vorhandenen Sicherheitspolitiken notwendig sind. Die präzise Identifikation von TCB-Funktionen erlaubt, die Implementierung der TCB-Funktionen von nicht-vertrauenswürdigen Systemkomponenten zu isolieren. Dadurch legen kausale TCBs die Grundlage für TCB-Implementierungen, deren Größe und Komplexität eine Analyse und Verifikation bezüglich ihrer Korrektheit und Unumgehbarkeit ermöglichen. Kausale TCBs haben ein breites Anwendungsspektrum – von eingebetteten Systemen über Politik gesteuerte Betriebssysteme bis hin zu Datenbankmanagementsystemen in großen Informationssystemen.

Acknowledgements

First of all, I would like to thank my supervisor Prof. Winfried E. Kühnhauser. He drew my interest to the exciting world of computer security research and supported me with lively and interesting discussions throughout my work. I also want to thank the reviewers of my dissertation for their valuable feedback and their preparedness to review my dissertation.

Special thanks to the Carl-Zeiss-Stiftung for awarding me a PhD scholarship and thus providing the financial background for my research.

Further, I would like to thank my colleagues for all the valuable and motivating discussions. This particularly concerns Peter Amthor, who was always prepared for in-depth discussions and valuable feedback over a cup of tea, and my favorite office mate Katja Hose, who is a good listener. I also want to thank the students and their theses that contributed to my research work with new ideas and their implementation. In particular, special thanks to Felix Neumann for having the guts and the endurance for dealing with formal specifications. I would also like to thank all my friends and neighbors for their continuous motivation and support as well as for their occasional pestering – all kept me going. Thanks to my friend Janine Seifert for a recreational girl's night out once in a while; I always had a good time.

Last but not least, I would like to thank my family for their moral support – my parents always made me believe that I can achieve my goals. I also like to thank Alex for his never-ending energy, administrative support, and confirmation.

Contents

1 Introduction

The man who asks a question is a fool for a minute,
the man who does not ask is a fool for life.

Confucius

IT systems with advanced security requirements increasingly apply problem-specific *security policies* – sets of rules designed to meet a system's security goals [73] – for describing, analyzing, and implementing security properties [46, 60, 74, 105, 165]. To precisely describe security policies, formal security models such as [26, 41, 74, 100, 108, 181, 212] are applied, allowing for formal analyses of security properties, and serving as specifications, from which policy implementations are generated [18, 65, 176, 180, 241].

Security policies are part of a system's *trusted computing base* (*TCB*). Consequently, correctness and tamper-proofness of a TCB's implementation are of paramount importance for establishing, preserving, and guaranteeing a system's security properties.

In the general use of the term, a TCB denotes all components of a system that implement its security properties. In this dissertation, we adopt a functional view and apply the term to those and only those system functions that are necessary and sufficient to establish, enforce, and preserve a system's security properties. Moreover, we distinguish between a TCB, its *security architecture*, and the *security architecture's implementation.* We apply the term security architecture to that part of a system architecture that implements the TCB; here, in contrast to general system architectures, additional requirements apply (such as the reference monitor principles [17, 97, 125]). The term security architecture's implementation (also called the *TCB's implementation*) then refers to the source code level, where data types and algorithms implement the TCB.

In general it holds that the lower the complexity of a TCB and the smaller its functional perimeter, the easier it is to assert the correctness and tamper-proofness of a TCB's implementation [97, 220]. The essential prerequisite for such an implementation is the precise identification of a TCB's functions. These serve as a specification for the functional requirements of a security architecture, from which an implementation is finally derived. This dissertation focuses on the starting point of these dependencies – the identification of the functional perimeter of TCBs.

1.1 Motivation

Implementing security policies is not an easy task. Today's commodity operating systems often have serious deficiencies in this regard; for more than forty years, they have provided only a rather elementary support for discretionary identity-based access control (IBAC) policies. As a consequence, the implementation of more sophisticated, application-specific security properties has moved to the application level. For example, Web browsers implement policies to prevent the execution of malicious code (e.g., by cross-site scripting), to identify phishing sites, or to protect children from harmful content. Corporations integrate enterprise policies that reflect their organizational hierarchies into Enterprise Resource Planning (ERP) systems, and e-mail clients implement mail integrity and confidentiality policies. As a consequence, major parts of the applications' policies are implemented by the applications themselves, resulting in a large, heterogeneous, and distributed TCB implementation, caused by a runaway evolution with independent contributions from many research groups and organizations. Thus, it is hard to precisely identify a TCB's functional perimeter, which renders essential TCB properties such as correctness, robustness, and tamper-proofness difficult to achieve.

On the other hand, imprecisely defined TCBs are also a problem of embedded systems, which have strict constraints with respect to their resources. That means, the larger the functional perimeter of the TCB of an embedded system, the larger is for instance its memory footprint, which may lead to violating the system's resource constraints.

This dissertation focuses on TCBs of operating systems. In this context, there have been many approaches that combine the policy components of operating systems and applications into a central component, thereby allowing for methodical policy engineering and analysis, and for isolating the policy components. Operating systems – which we refer to as *policy-controlled operating systems* in the following – now provide kernel abstractions for security policies along with a policy runtime environment to protect and enforce the security policies, e.g., *Security Enhanced Linux* [165, 166] or *Security Enhanced BSD* [197, 240]. Applications are now able to integrate their individual security policies into the kernel. As a result, the TCB implementations of these systems are less distributed, less heterogeneous, hence easier to identify and isolate. However, they are neither smaller nor less complex.

The reasons for the large size and complexity of the TCBs of policy-controlled operating systems are twofold. Policy-controlled operating systems are based on commodity operating systems, thereby maintaining their monolithic architectures and scattering their policy enforcement mechanisms all over the kernel. On top of that, the new kernel abstractions of policy-controlled operating systems share the ambition to be universal. In order to provide runtime support for a wide variety of security policies, the policy decision environment as well as the policy enforcement mechanisms are designed in an all-round fashion, rendering them large, complex, and expensive.

Nevertheless, the approach of combining security policies into a central system component is a major step towards methodical policy engineering and well-designed, analyzable, and tamper-proof TCBs. To deal with the deficiencies of policy-controlled operating systems, the dissertation develops an approach for methodical TCB engineering. This allows for precisely identifying a TCB's functional perimeter by systematically deriving the functions of a TCB from its security policies. The goal is to set the course for a TCB's implementation whose small size and complexity reduce the effort of analyzing and verifying a TCB's correctness and tamper-proofness.

1.2 Causal Trusted Computing Bases

The approach of this dissertation is to tailor a TCB's policy decision and enforcement environment to support only those security policies that are actually present in a TCB. A TCB's functional perimeter is then determined by exploiting causal dependencies between security policies and TCB functions. This approach results in policy-specific *causal TCBs* that contain only those functions that are required to establish, enforce, and protect the present security policies.

The general idea of tailoring TCBs to specific application scenarios is not new. Kernelized security architectures such as Nizza [122, 123, 220] proved application-specific TCBs to be feasible, and this is fundamental for this dissertation. We develop a systematic and policy-independent causal TCB engineering method that formalizes causal dependencies between security policies and TCB functions. This general method then leads to increasing the efficiency and effectiveness of TCB engineering due to reusability of TCB functions and policy-independent tool support.

1.3 Challenges and Contributions

We discuss the challenges and contributions of this dissertation along three questions:

- What is a suitable starting point to derive TCB functions from security policies?
- What is a suitable methodical approach to derive TCB functions from security policies?
- How can we implement the derived TCB functions in a security architecture?

The first question focuses on the formal representation of a TCB's security policies and imposes the challenge to *identify a uniform model notation.* Security policies are usually described by informal sets of rules. Since these are inadequate for policy analyses and policy implementation, the security community has developed numerous formal security models such as [26, 41, 74, 100, 108, 181, 212, 251, 254], which allow for formal analyses of security properties and increasingly serve as specifications for policy implementations. This dissertation shows that formal security models are also suitable to serve as a basis for specifying the functional requirements of causal TCBs. However, the wide variety of models makes it difficult to exploit common abstractions for causal TCB engineering. Based on the shared model abstractions of existing security models, we show that a uniform model notation that can express a wide variety of significant models can be found, which serves as basis to formalize security models, in order to derive TCB functions.

This dissertation answers this challenge by making the following contributions:

1. Identification of shared model abstractions of access control and information flow security models.
2. A uniform model notation for access control and information flow models.
3. An engineering method for access control and information flow models in the uniform model notation that allows for model-independent methods and tools for causal TCB engineering.

The second question is related to a systematic and model-independent engineering approach to identify a causal TCB's functional perimeter. This imposes the challenges to *create a functional design for causal TCBs*, to *develop an approach that derives TCB functions from a TCB's policies*, and to *specify a rule system that formalizes causal dependencies between policies and TCB functions.* The functional design specifies a framework for the derived TCB functions. It must consider that policies with shared model abstractions also have identical requirements regarding the functional perimeter of a TCB. On the other hand, policy-specific abstractions result in policy-specific requirements and thus in policy-specific TCB functions. In addition, all policies require TCB functions to ensure that they cannot be bypassed and are protected against unauthorized manipulations. Thus, a functional TCB design must contain a policy-dependent and a policy-independent component. Moreover, it has to meet specific requirements such as support for policy substitution, multiple policies, and engineering methods and tools. In particular, the causal TCB engineering method that derives TCB functions from a TCB's policies needs to be supported. The main requirement for this method is to produce a functional perimeter of TCBs that is nonredundant and complete with respect to a TCB's policies. Therefor, a sound theoretical foundation, including a rule system to formalize causal dependencies, is required, which also provides the basis for an automated TCB composition tool.

These challenges are addressed by this dissertation through the following contributions:

4. A functional design for causal TCBs that consists of a policy-independent and a policy-dependent TCB component.

5. A rule system that formalizes causal dependencies between security policies and TCB functions.

6. A model-independent causal TCB engineering method that precisely identifies the functional perimeter of
   - the policy-independent TCB component and
   - a TCB's policy-dependent component based on the formal rule system.

7. A sound theoretic and methodic foundation that allows for an automated TCB composition tool.

The last question deals with implementing the functions of causal TCBs. While the implementation of the policy-independent TCB component is to be accomplished once, implementing the policy-dependent TCB component is a repeating process for each individual application scenario. This imposes the challenge *to provide policy-independent method and tool support* that generates a TCB's implementation from its functions. Here, the main requirement is that the completeness and consistency of the resulting TCB implementation with respect to the identified TCB functions are ensured.

The dissertation's contributions with respect to this challenge are:

8. A policy-independent formal specification method that provides the basis for automated TCB generation tools.

9. Formal TCB specifications that are amenable to tool-supported validation techniques and that provide a sound foundation for automated source code generation tools.

1.4 Organization

This dissertation is divided into eight sections. Section 2 discusses related work on the policy runtime environments of policy-controlled operating systems and approaches that aim at reducing the size and complexity of a TCB's implementation. It also surveys access control and information flow models that are significant for modeling real-world security policies. Thereby, relevant terminology is introduced.

Based on this survey, Section 3 introduces a uniform model notation. Additionally, it presents an engineering method to develop models in the uniform notation, demonstrates it by means of examples, and evaluates it regarding its expressive power and the involved modeling costs. In doing so, this section also introduces an RBAC policy that is based on a real-world application scenario and serves as running example. Moreover, this section discusses related work on the uniform model notation.

Afterwards, Section 4 presents a functional causal TCB design along with a systematic causal TCB engineering method. Thereby, this section motivates the functional perimeter of a TCB's policy-independent component and presents a model-independent engineering method that, based on a formal rule system, derives the functions of the policy-dependent TCB component. The latter is demonstrated by means of the RBAC policy.

To implement a TCB's policy-dependent component without jeopardizing the implementation's correctness and completeness with respect to the identified TCB functions, Section 5 presents a formal TCB specification method. Therefor, it discusses requirements for TCB specifications and presents approaches to meet them. A formal TCB specification for the RBAC policy is presented afterwards.

Section 6 demonstrates that the main objective of this dissertation – the precise identification of a TCB's functional perimeter – has been achieved. In addition, it evaluates the feasibility of the model-based engineering approach of causal TCBs with respect to its application in real-world scenarios.

Finally, Sections 7 and 8 conclude this dissertation with a summary and a discussion of future work.

2 Related Work

One's person paranoia is another person's engineering redundancy.

Marcus J. Ranum

The goal of this section is to review related work that is relevant to the goals and the approaches of this dissertation. On this account, we present work in three research areas: work on policy-controlled operating systems, TCBs, and security models for access control and information flow policies. Section 2.1 presents policy-controlled operating systems with the objective to discuss the properties of their TCBs. Here, we focus on the design of the systems rather than their implementation. In the area of TCBs, we present approaches that aim at reducing the size and complexity of a TCB (Section 2.2). Afterwards, Section 2.3 discusses security models in with the goal to identify common model abstractions in the following.

2.1 Policy-controlled Operating Systems

The objective of policy-controlled operating systems is to support a high degree of security policy flexibility, which comprises a wide variety of security policies that may even change for instance if new application software or a system update is installed. This is attained by kernel abstractions for security policies along with a policy runtime environment (RTE) that strictly isolates policy enforcement mechanisms – generally called *Policy Enforcement Points (PEPs)* – from the policy decision environment (*Policy Decision Point (PDP)*). That way, different policies can be integrated in a system's PDP and policy flexibility is provided by adapting the PDP without modifying the PEPs.

Policy-controlled operating systems have been developed for almost 20 years. Prominent representatives are the *Flask* operating system (OS) [225], *Rule Set Based Access Control (RSBAC)* [189,191], and the *Policy Machine* [82,84,115]. Particularly based on the Flask security architecture introduced by the Flask OS, further reference implementations for various platforms and application scenarios have been developed. This section presents these systems with the objective to discuss the properties of their TCBs. While Section 2.1.1 introduces the Flask security architecture along with prominent reference implementations, Sections 2.1.2 and 2.1.3 present RSBAC and the Policy Machine.

2.1.1 Flask Security Architecture

The *Flask* security architecture [225] was first implemented in the Flask OS, which is based on the Fluke microkernel [93]. The main goal of the Flask architecture is to provide for policy flexibility by ensuring that all its components always have a consistent view of policy decisions; secondary goals are application transparency and a low performance impact [225]. Based on these goals, the components of Flask are a set of *objects managers* and a central *security server* (Figure 2.1). The security server is the PDP of Flask and contains the system's security policy. Object managers are responsible for enforcing policy decisions and thus are PEPs. That way, when a process tries to access an object, e.g., a file or another process, the responsible object manager traps this access, queries the security server, and enforces its decision by either granting or denying the access. To minimize the performance overhead, all object managers contain an *access vector cache (AVC)* that allows them to cache the access decisions of the security server [225].

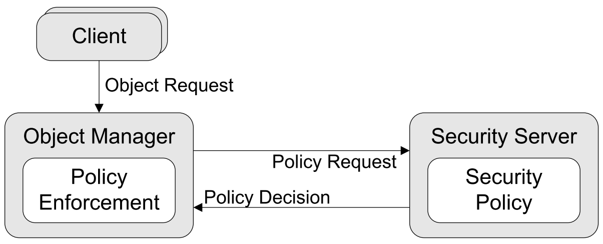

Figure 2.1: The Flask Security Architecture [225]

Besides requesting policy decisions, object managers also request labeling decisions from the security server. Labeling decisions specify security attributes that each policy-controlled object is assigned to. Security attributes are subsumed in a policy-specific *security context* that may consist of a user identity, a classification, or a role [225]. However, since the performance impact would be too high if labeling and policy decisions were based on this context, each object is also given a policy-independent *security identifier* [225]. The security identifier is created by the security server based on the policy by determining a context for an object and deriving a corresponding identifier. The identifier is given to the object manager, which binds it to the new object. The Flask OS implements four types of policies: multi level security (MLS), type enforcement (TE), IBAC, and role-based access control (RBAC).

Based on this security architecture, several reference implementations have been developed; most prominent ones are *Security Enhanced (SE) Linux* [165, 166] and *SEBSD* [197, 240]. SELinux [165, 166] implements Flask as part of the Linux kernel with the goal to address the weaknesses of the Linux' discretionary access control (DAC) mechanisms [165]. The SELinux security server therefor defines a mandatory access control (MAC) policy that is a combination of TE, RBAC, and optionally MLS. The security context thus contains a user identity, a role, a type, and optionally an MLS level/range. SELinux originally provides object managers only for kernel objects. As a consequence, application-specific objects are not controlled by the policy. Recent work such as [52, 133, 244] has tackled this problem by developing userspace object managers that enable control for such objects. For example, [52] has developed a userspace object manager for GConf [194], a configuration application for the GNOME desktop. In doing so, the flexibility of the SELinux policy is further improved. On the other hand, userspace object managers contribute to distributing the TCB on kernel and application level. As a consequence, essential TCB properties such as correctness, robustness, and tamper-proofness are difficult to achieve.

SEBSD [197, 240] is a port of SELinux to FreeBSD in the context of the TrustedBSD project [197]. Here, the goal is to run the Flask architecture as a plug-in module to the TrustedBSD MAC Framework [246]. Since SEBSD is a port of SELinux, the policy used by SEBSD is roughly the same as the SELinux policy; differences result only from system adaptations [240]. The SELinux AVC and security server are also directly ported [240]. Thus, SEBSD and SELinux mainly differ in their implementations.

Convinced by the feasibility of the existing Flask reference implementations, Flask has also been implemented in the Android OS for mobile devices, e.g., *SE Android* [217, 222], *FlaskDroid* [47–49], or [56, 175]. The goal of SE Android is to address the weaknesses of DAC that occur since Android uses Linux access control mechanisms [222] by integrating the SELinux implementation in Android. This leads to three main benefits: (i) confinement of privileged Android processes, (ii) isolation of Android apps, and (iii) a centralized policy that can be engineered and analyzed [222]. The migration of SELinux to Android has imposed many challenges, one of which is the specification of a suitable policy. Due to a different application scenario of Android compared to a conventional Linux system and the large size of the SELinux reference policy, the latter is not suited for mobile devices. Thus, a specialized policy has been developed, which combines TE and MLS to confine domains for system processes and user apps, and to isolate apps and files from each other. The result is that the SE Android policy is much smaller in terms of the binary policy size, the number of domains, objects, allow rules, and type transitions [222]. As with SELinux, SE Android also supports userspace object managers that allow for applying policy control on application-specific objects [222]. Thereby, the TCB is enlarged and distributed, hence harder to identify and isolate.

Similar to SE Android in goals and approaches are *TrustDroid* [48] and *XManDroid* [47], which are subsumed by *FlaskDroid* [49]. The objective of FlaskDroid is to provide a generic security architecture for the Android OS that can serve as a flexible and effective framework for different security requirements [49]. The approach is to provide a set of object managers both on kernel level and on userspace level. The latter enforces access control decisions made by a userspace security server, while kernel-level object managers request policy decisions from a kernel-level security server. The policies of both servers are synchronized at runtime; for instance a change in the userspace policy must be supported by the kernel-level policy [49]. As a result, FlaskDroid can provide a fine-grained and highly flexible access control security policy. On the other hand, the TCB is enlarged and is distributed on userspace and kernel level. On top of that, the policy is also distributed, which complicates security policy engineering and analysis.

2.1.2 Rule Set Based Access Control

Rule Set Based Access Control (RSBAC) [3, 189–191] is an orthogonal approach to Flask that aims at providing support for mandatory and fine-grained access control. Based on the Generalized Framework for Access Control (GFAC) [4, 148], RSBAC extends the Linux kernel by a security architecture that consists of three components (Figure 2.2): the *Access Decision Control Facility (ADF)* along with the *Access Control Information (ACI)* component form the PDP, while the *Access Enforcement Facility (AEF)* represents the PEP. The ADF enforces a set of different policies, including instances of the Bell-LaPadula model, simple RBAC policies, or a malware scanner policy; for more details refer to [190, 191]. Besides, the ADF also implements a meta policy that combines the results of all policies that are involved in policy decision making. The ACI module is responsible for administrating the security attributes of objects and subjects [191], including attribute assignments as well as the nonvolatile attribute storage of nonvolatile objects.

In summary, RSBAC mainly differs from the other systems in the supported security policies and implementation details. This includes for example a missing cache for policy decisions in the PEPs, missing support for userspace PEPs, or the usage of Linux user identifiers instead of a platform-independent security identifier. The latter particularly leads

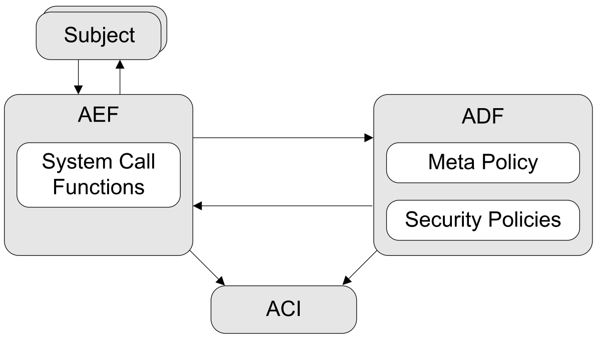

Figure 2.2: The RSBAC Security Architecture [191]

to that RSBAC always depends on Linux user management such that its TCB can not be properly isolated, even though Linux were not based on a monolithic architecture.

2.1.3 Policy Machine

The *Policy Machine (PM)* [61,82,84,89,115] is a security architecture with the goal to provide a unifying framework for a wide range of attribute-based access control (ABAC) policies or policy combinations through a single mechanism. The PM can be implemented in both centralized and distributed systems. For this reason, it consists of a *PM Server*, which acts as PDP, including a PM data base, and a Policy Administration Point (PAP), and a set of *PM Clients*, each comprising a PEP, an application programming interface (API), and PM-aware applications. Figure 2.3 shows the system architecture. The foundation of the PM

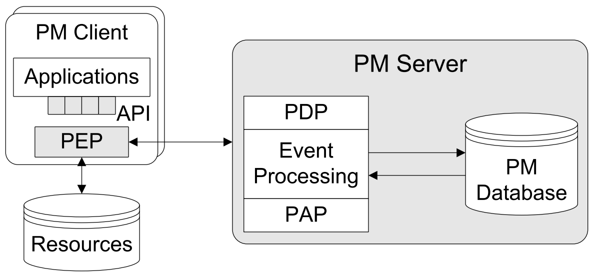

Figure 2.3: The Security Architecture of the Policy Machine [82]

server is the PM access control framework, which provides a uniform formalism for a range of access control policies based on shared model abstractions; details on this formalism are provided in Section 3.6. To implement the policies, the PM server, which also is referred to as “logical machine”, contains a fixed set of data relations and functions [82]. By managing the data relations via the PAP, the PM server is configured to support specific policies. The functions of the PM server aid in policy decision making and thus enforcing the expressed policies. Thereby, the event processing module may dynamically update the configuration of the policies by client events. The authors have developed a reference implementation of the PM server that can enforce a wide variety of security policies, including instances and combinations of DAC, MAC, RBAC, ABAC, and Chinese Wall; for more details refer to [61, 82].

The PEP within the client controls each access attempt and requests a policy decision from the PDP within the PM server to allow or deny the access request [82]. The physical location of any object that a client may access is only known to the PM server. Thus, in case an access is allowed by the PM server, the policy decision response is accompanied by the object's location such that the client can enforce the policy decision.

The PM is the work that is most closely related to this dissertation. It follows the same approach by identifying a uniform security model notation and developing a corresponding policy RTE, which is able to enforce a wide variety of access control models that express both operating system policies as well as application policies. However, the PM is a "general purpose protection machine" [82], which offers a large library of supported security policies. Thus, the TCB of the PM also is a general purpose one that contains more functions than actually required for a specific application scenario. The authors tackle the problem of a large TCB at implementation level by decoupling policy decision making from the host system.

2.1.4 Summary

All discussed policy-controlled systems have in common that they follow a similar approach by strictly isolating the PDPs from PEPs in order to provide policy flexibility; however, they differ greatly in their implementations and the types of security policies they support. Compared to the contributions of this dissertation, the main goal of these systems is policy flexibility and not the reduction of the size and complexity of their TCBs. Though indeed some of them are concerned about the amount of trusted code, reducing the latter is only a subordinated goal and is approached at the level of implementation. Besides, all systems aim at general-purpose PEPs and PDPs, resulting in large and complex TCBs that contain more functions than actually required for a specific application scenario. On top of that, by allowing for userspace PEPs or even userspace PDPs, the policy components are not located exclusively in the kernel anymore. As a result, the TCBs are again distributed, which makes their identification and isolation hard, rendering essential TCB properties such as correctness, robustness, and tamper-proofness difficult to achieve. Additionally, userspace PDPs also result in decentralized security policies, which complicates policy engineering and analysis in order to establish a system's security properties.

2.2 Reducing the Size and Complexity of TCBs

The goal of this section is to discuss related work that aims at reducing the size and complexity of a system's TCB. All presented work has in common that it tackles the problem of TCB reduction at the level of TCB implementation rather than at the functional level like this dissertation does. Among this work, one approach – *Nizza* [122, 123, 220] – stands out since it not only is groundbreaking for our approach to engineer policy-specific TCBs but also provides a sound implementation platform for causal TCBs. On this account, Section 2.2.1 introduces Nizza in more detail. Afterwards, Section 2.2.2 summarizes other approaches.

2.2.1 Nizza Security Architecture

Nizza [122, 123, 220] is a security architecture that aims at reducing a system's TCB by tailoring it to a specific application scenario. It is based on two main requirements – a trusted software stack and compatibility with legacy software [123] –, which have lead to three fundamental design principles. (i) Security-sensitive source code is strictly isolated from security-insensitive code, thereby reducing the TCB implementation. (ii) Nizza uses *trusted wrappers* to reuse untrusted system components such as device drivers, or file systems, in order to provide confidentiality and integrity [123]. (iii) A container is used to run an untrusted standard OS isolated from security-sensitive components.

Based on these design principles, a security architecture is derived that consists of four parts: A small kernel, a secure-platform layer providing trusted service, secure applications, and an untrusted legacy OS [123]. The reference implementation of Nizza is based on the

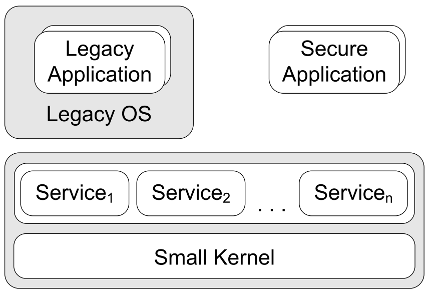

Figure 2.4: The Nizza Security Architecture [123]

Fiasco microkernel [112], which is responsible for enforcing domain isolation and for allowing communication between the domains. To provide the full functionality of a standard OS, the reference implementation uses L^{4}Linux, which is a paravirtualized Linux kernel that uses the L4 microkernel interface instead of directly accessing the hardware [123]. Hence, L^{4}Linux is executed in untrusted mode and cannot corrupt other system components. The secure-platform layer is application-specific and contains only those trusted services that are fundamental for an application scenario. Such services are typically a trusted loader, a trusted GUI, or secure storage. Secure applications are those application parts that execute security-sensitive code. This code is isolated from the remaining, i.e., security-insensitive, application part and executed within the TCB. Identifying the point of separation of security-relevant and security-irrelevant source code along with defining suitable interfaces is done manually and has to be accomplished for each software application individually. From a TCB engineering point of view, this is an obstacle to engineering efficiency (e.g., by reusing code) and automated TCB composition.

In this dissertation, we take Nizza's approach one step further by developing a systematic and application-independent causal TCB engineering method. This leads to increasing the efficiency and effectiveness of TCB engineering due to reusability of TCB functions and policy-independent tool support. Besides, the Nizza reference implementation provides a sound implementation platform for causal TCBs. The reasons are as follows: (i) it already implements functions that are required by causal TCBs, e.g., isolation mechanisms, authenticated booting, a trusted GUI, and secure storage. (ii) The implementation of these functions is demonstrably small [123]. For example, the TCB of a Nizza implementation for signing emails has about 105,000 lines of code (LOC), where a Linux 2.4 kernel with a minimal configuration already has 155,000 LOC [123]. (iii) Nizza is able to integrate the policy RTE of causal TCBs in a straightforward way – as trusted service within the secure-platform layer, and policies are isolated from each other by isolation mechanisms provided by the microkernel. By this means, Nizza can be enhanced by a tailored policy RTE that enforces security policies to control either the legacy OS and its legacy applications or any secure application.

2.2.2 Other Approaches

A research project that is closely related to Nizza is *Mikro-SINA* [110], which aims at reducing the size and complexity of the TCB of a Virtual Private Network (VPN) gateway. The approach is based on the Nizza security architecture; thereby extracting the security-relevant functions of a VPN implementation and executing them in a separate protection domain. The result is that the size and complexity of the TCB in means of LOC is reduced by an order of magnitude compared to using a standard Linux kernel [110].

A security architecture that basically shares its goals with Nizza but is an orthogonal approach is *Perseus* [50, 195, 201]. It executes secure applications on a minimal security platform and provides support for a legacy OS, in order to provide compatibility for standard security-insensitive application software. Thus, Perseus and Nizza are very similar; the Perseus reference implementation even applies some of components used by the Nizza reference implementation such as Fiasco and L^{4}Linux.

The goal of the *VPFS* project [249] is to provide a secure and reliable storage for security-sensitive applications that run on top of a microkernel, which may concurrently execute untrusted application software. This is achieved by a security architecture for a virtual private file system (VPFS) that is based on both a small amount of trusted storage and an untrusted legacy file system residing on the same machine.

In general, a small TCB implementation with low complexity sets the basis for formally verifying a TCB's correctness and tamper-proofness. This statement is substantiated by the *seL4* microkernel [137, 138], which is a formally verified microkernel based on interactive, machine-assisted, and machine-check proof techniques. Thereby, [137,138] have shown that a formal verification is practically achievable for OS microkernels. As a consequence, kernelized application-specific TCBs such as the ones discussed before not only are feasible with respect to their implementation, but also provide the basis for verifying a system's security properties.

In contrast to approaches that build kernelized security architectures, other research work such as [53, 109, 143, 170] aims at reducing the size and complexity of the TCBs of todays commodity, i.e., monolithic, operating systems. For example, [143] reduces the TCB of Linux systems by analyzing them with respect to unused kernel source code and removing it. By this means, the general-purpose Linux OS also is to a certain extend tailored to a specific application scenario. On the other hand, *Flicker* [170] is an infrastructure for executing security-sensitive source code in complete isolation alongside a commodity OS. Flicker therefor utilizes hardware support for late launch and attestation, which has already been introduced in commodity processors from AMD and Intel [170].

Finally, the motivation of a small TCB with low complexity is also shared by virtualization systems. This has lead to several research projects that aim at developing hypervisors with small TCBs, e.g., *NOVA* [227,238], *TrustVisor* [169,171], *SecVisor* [216], or *Terra* [96].

2.3 Security Models

The goal of this section is to survey existing security models, in order to identify common model abstractions in the following. Based on the latter, we then specify a uniform model notation, which provides the basis for model-based causal TCB engineering. The secondary goal of this section is to discuss how causal TCB engineering improves the efficiency and effectiveness of model-based security policy engineering. On this account, Section 2.3.1 discusses the process model of model-based security policy engineering that this dissertation refers to and thereby provides the terminology used. Afterwards, Sections 2.3.2 and 2.3.3 survey existing security models of the access control and information flow control domains.

2.3.1 Model-based Security Policy Engineering

Causal TCB engineering improves the efficiency and effectiveness of model-based security policy engineering, since it can be integrated without semantic gaps and is supported by policy-independent tools. This claim is supported by this dissertation by showing that (i) causal TCB engineering uses existing security models that are rewritten in a uniform model notation, (ii) causal dependencies between security policies and TCB functions can be formalized, which allows for tool-supported TCB composition, and (iii) a specification method can be developed, which enables automated source code generation of a TCB's implementation. On top of that, security policies that are expressed by models in a uniform model notation are unlocked to a family of model-independent analyzing methods and tools [15, 16, 91, 144].

The resulting security policy engineering process consists of six steps (Figure 2.5). At the

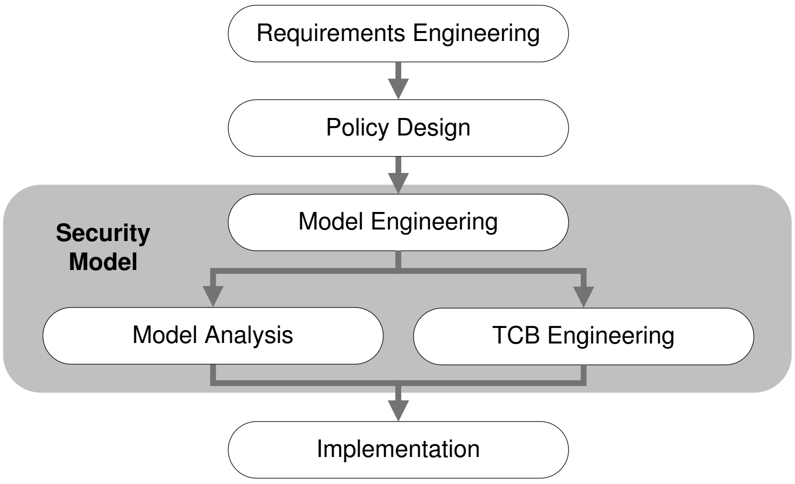

Figure 2.5: Model-based Security Policy Engineering

beginning, informal security requirements are stated as precisely as possible by a thorough requirements engineering. In this context, several specialized approaches for model-based requirements engineering have emerged, e.g., [78, 80, 124, 135]. In the next step, a set of operational rules is designed to meet the identified security requirements [73], resulting in an informal security policy, e.g., [77, 98, 168, 231], that suits the specific needs of the intended application scenario. The policy is then expressed by formal means to get a sound foundation for policy analysis or TCB engineering. In doing so, security model engineering either uses an existing domain-specific *security model* or creates a new one for instance by using a uniform model notation. A security model is a set of abstractions to precisely describe the semantics of a policy such as object classification, role-based access rules, or auditing and logging practices. Once formalized in a suitable model, the policy is analyzed by methods and tools that operate on this model or the uniform model notation such as [16]. During this step, the formalized policy – which we refer to as *model instance* of this particular model – is analyzed with respect to the required security properties, e.g., access right proliferation [15, 91, 106, 209], information flow leakage [67, 104, 218, 242], or implementation adequacy [28, 252]. These are derived from an application's security requirements and are formally defined by its security model. That way, a valid formal representation of a policy is provided. Parallel to model analysis, the functional perimeter of the TCB that is required to enforce and protect the policy is engineered. Here, TCB functions are derived from a model in a uniform model notation, in order to compose the functional perimeter of a causal TCB. Both, the formal model instance as well as the derived TCB functions, then provide a precise specification for a policy's implementation.

2.3.2 Access Control Models

Butler Lampson published the ground-breaking work of access control models in 1971 [145]. The goal of his model is to provide precise rules about which subjects in a computer system (users or processes) are allowed to execute specific operations (such as read or write) on the objects (such as files or printers) in that system. Lampson used a simple *access control function (ACF): subjects* $\times$ *objects* $\times$ *operations* $\rightarrow \{true, false\}$ (often represented by an isomorphic access control matrix) to model a snapshot of the protection state of an access control system.

Lampson's model was augmented by a state automaton with the objective to model dynamic behavior of the protection state [69, 101]. Each state of the automaton is a triple (S, O, m) where S and O are sets of subjects and objects respectively, and $m : S \times O \rightarrow 2^{rights}$ is an access control matrix (ACM). S, O, and m can now be modified by rules that create and destroy subjects/objects, as well as grant/transfer and delete access rights, which represents the automaton's state transitions. By combining an automaton with ACMs, model dynamics was introduced; this paved the way for modeling real-world security policies, for which modifying privileges and adding/deleting users and objects are important features.

Harrison, Ruzzo, and Ullman [107] combined ACMs and state machines more formally. Each state of an HRU model reflects a single protection state of an access control system; state transitions are triggered by system-specific operations that modify this state. Security properties such as right proliferation now can be analyzed by observing state transitions caused by input sequences; in particular, the boundaries of right proliferation can be explored by state reachability analyses. Formally, any HRU security model is a state machine (Q, Σ, δ, q_0) with a state set Q, an input alphabet Σ, a state transition function δ, and an initial state q_0. Each state $q \in Q$ is a triple (S_q, O_q, m_q) where S_q and O_q are the subject and object set of that state, and $m_q : S_q \times O_q \rightarrow 2^R$ is the state's ACM with a finite right set R. Model dynamics are defined by the transition function $\delta : Q \times \Sigma \rightarrow Q$ that models the impact of application-specific operations on the state. In real-world applications, operations in Σ typically (but not exclusively) are used by security administrators and eventually involve users, files, or access rights.

HRU models allow for the analysis of the dynamic behavior of access control systems. In particular, their property whether to leak access rights into matrix cells was addressed as the HRU safety problem, which is not decidable in general. As a consequence, several safety-decidable fragments of the HRU calculus emerged, which buy safety decidability by limiting the expressive power of the calculus, e.g., *monotonic* HRU models [106], or by introducing news abstractions like *types*, e.g., the *Schematic Protection Model (SPM)* [12, 13, 204] or the *Typed Access Matrix (TAM)* model [209].

The above mentioned access control models are mainly applied to model IBAC policies. In the early 1990s however, IBAC policies became insufficient as commercial applications required application-oriented access control policies that were in compliance with business processes. Hence, the ability to specify enterprise-specific access control policies and to streamline the typically burdensome process of authorization management have been the main motivations for RBAC models [86, 87, 205, 212] and ABAC models such as [39, 63, 130, 147, 219, 251, 254]. The central idea of RBAC models is that rights are associated with roles, which are assigned to users. While roles reflect various job functions, users are assigned to roles based on their responsibilities and qualifications [210]. The ACM is thus defined as $m : roles \times objects \rightarrow 2^{rights}$. Analogous to IBAC models, models such as *URA97 RBAC* [129], *miniARBAC* [229], *PARBAC* [228], or the RBAC model of [157] then combine this ACM with an automaton to express model dynamics. Besides, other approaches combine the ACM with additional security-relevant attributes such as spatial attributes [31, 64] or temporal attributes [30, 131].

The main application scenarios of ABAC policies are web-based applications like web services where user attributes such as age or location are of more importance than their identities. Access control rules are based on these attributes and the ACM is defined as $m : subjectattributes \times objectattributes \rightarrow 2^{rights}$. Like RBAC models, ABAC models are originally static snapshots of a protection system, e.g., [37, 39, 63, 147, 219, 251]. To model policy dynamics, attribute-based ACMs are combined with the notions of HRU models, resulting for instance in the *Attribute-based Access Matrix (ABAM)* model [254] or [59, 130, 254]. A state of the ABAM model is a tuple $q = (S_q, O_q, m_q, ATT_q)$ where S, O, and m are the standard HRU model components and ATT is the set of attribute tuple values that each subject and object is assigned to.

2.3.3 Information Flow Control Models

Access control models are designed to control immediate access to objects without taking into consideration information flow paths that are implied by a collection of access control rights [67]. The issue of unauthorized access by obscure information flow paths is tackled by information flow control models, which are primarily concerned with preserving confidentiality of sensitive information; only few models deal with information integrity.

The concept of information flow control was first formally defined by Denning's lattice model [67]. Since this model has provided the foundation for numerous follow-up work, we consider it the pioneer in the domain of information flow control models. The lattice model is a tuple $(S, O, L, cl, \oplus)$ where S and O are sets of subjects/objects, $L = (SC, \leqslant)$ is a lattice with SC being a set of security classes corresponding to disjoint classes of information and $\leqslant$ defining a flow relation on pairs of security classes (partial ordering). cl is a classification function that assigns a security class to each subject and object, and $\oplus$ is a class-combining operator that specifies an associative and commutative binary operator for pairs of security classes [67]. Information flow between subjects and objects is then controlled by information flow between their security classes (also called *security labels*) and the lattice's flow relation, which defines all authorized information flow paths between security classes.

A special case of information flow control policies are MLS policies. MLS policies are based on a hierarchy of security classes by a total ordering of the flow relation $\leqslant$ (also called *dominance relation*); and information flow is only authorized from a lower security level to a higher security level. The first formalization of MLS policies was published by Bell and LaPadula [26, 27] and is known as the *Bell-LaPadula (BLP)* (confidentiality) model. The BLP calculus combines a lattice (reflecting the confidentiality hierarchy) with the HRU calculus. In doing so, authorized information flow is modeled on an application-oriented level of abstraction via an *information flow graph*, and precise rules (known as the *Simple-Security Rule* and the $\star-$*Property*) govern the correct mapping of the information flow graph to the model's ACM (*BLP security*).

The work of Denning, Bell, and LaPadula has inspired many derivatives of these models. For example, Foley introduced a similar model (*Confinement Flow Model (CFM)* [92]) with the objective to allow greater flexibility in the assignment of security classes to subjects/objects, and to provide a model for a framework of security policies. This is done by adding a *confine*-function, associating a pair of security classes with each subject/object similar to [68,167]. Analogous to the BLP model, [92] applies an HRU derivate to model policy dynamics and implemented the lattice by a modified ACM.

A more popular derivative was motivated by commercial applications with the objective to protect integrity – the *Biba Integrity* model [35]. Instead of BLP's confidentiality hierarchy, the Biba model has an integrity level hierarchy, and information cannot flow up towards higher integrity levels. Correspondingly, the ACM controlling rules are dual to the BLP rules and are formulated in terms of No Read Down and No Write Up rules.

Another well-known security policy motivated by the commercial sector is the Chinese Wall policy, preventing unauthorized information flow between consultants and companies of the same line of business. A Chinese Wall model was published first by Brewer and Nash [41]. The Brewer/Nash model is based on ACMs (reflecting the discrete IBAC component) and relations reflecting access histories and rivalry between companies. Later, an MLS model for enforcing this policy was introduced by Sandhu [207, 208], in which mutually disjoint conflict of interest (COI) classes control information flow between companies and consultants. Each object of the model is labeled by security labels that are defined as n-element vector $[i_1, i_2, ..., i_n]$ where each $i_k \in COI$. The model also uses the *Simple Security Rule* and the $\star-$*Property* from BLP-Models to govern the correct mapping of the information flow graph to the ACM.

A more recent application scenario for MLS models is dynamic web service composition. Here, complex services that are available on the Web are composed at design time by service composition planners such as SHOP2 [118]. Hutter et al. [118–120] have developed an MLS model and a system implementing this model for automatically composing web services for collaborative business processes and health care composition plans. Compared to the BLP model, the difference is that the security level of a web service is subdivided with respect to the type of information. Thus, depending on a specific model instance, different lattices are combined, i.e., each lattice $(C_1, \leqslant_1), (C_2, \leqslant_2), \ldots, (C_n, \leqslant_n)$ represents one type of information, to a composed lattice $L = (C, \leqslant)$ by $C = C_1 \times C_2 \times \ldots \times C_n$ and $(c_1, c_2, \ldots, c_n) \leqslant (c'_1, c'_2, \ldots, c'_n)$ if $c_i \leqslant c'_i$ holds for all $1 \leqslant i \leqslant n$.

Parallel to researching static and dynamic lattice-based models for information flow policies, various other approaches have emerged such as programming-language techniques for specifying and enforcing information-flow policies [200], decentralized information flow models like the *Decentralized Label Model (DLM)* [172–174], or models to specify information flow policies that may change during policy runtime, e.g., *Flow Locks* [42–44]. These models are out of the scope of this dissertation since they use other levels of abstractions for modeling information flow policies.

2.4 Summary

This section has discussed relevant related work in the research areas of policy-controlled operating systems, TCBs, and security models. In doing so, it has shown that today's policy-controlled operating systems have large and complex TCBs. Thus, it is hard to precisely identify the TCBs' functional perimeter. Furthermore, this section has introduced approaches to develop small TCBs. All approaches have in common that they tackle the problem of TCB reduction at the level of TCB implementation. As a consequence, the great potential of reducing the size and complexity of TCBs already at higher levels of abstraction, i.e., during earlier steps of software development, is not exploited. Finally, this section has presented a wide variety of security models in the domain of access control and information flow control. Thereby, it has shown that many of these models share commonalities.

The composition of these facts leads to the methodical approach of this dissertation.

3 Security Model Core

It is a capital mistake to theorize before one has data.
Insensibly one begins to twist facts to suit theories,
instead of theories to suit facts.

Sir Arthur Conan Doyle,
Sherlock Holmes

Section 2.3 has shown that diverse security models have been developed, which makes it difficult to exploit common abstractions for model engineering, model analysis, or model implementation. Our goal is to provide the foundation for these activities by analyzing a considerable amount of security models – namely these, which have either been proven relevant in practice or provided the foundation for further security models – in order to reveal common model abstractions. The approach is to identify a uniform model foundation that is common to many security models allowing for model-independent engineering methods and engineering tools. We will refer to this uniform model foundation as *security model core* in the following.

The model core is a key enabler for engineering models with shared model abstractions and thus for engineering common policy specifications. This enables us (i) to facilitate the process of model engineering by reusing model components, (ii) to develop model-independent methods for model analysis and implementation, (iii) to provide model-independent tool support for model engineering, model analysis, and model implementation, and (iv) to develop causal TCBs that are tailored to a specific security model. Out of these four merits, this dissertation focuses on methods for developing causal TCBs.

The goal of this chapter is to show that many security models share such a common model core. For this purpose, we identify common model components by showing that models emerge from each other and form a family tree (Section 3.1). On this account, the common model core is exposed for which we afterwards provide a formal definition (Section 3.2). The formal model core along with a core-based model engineering method then establishes the basis for designing causal TCBs. For this purpose, Section 3.3 proposes core-based model engineering. Afterwards, we evaluate the feasibility and expressive power of the model core along with core-based model engineering by rewriting three well-known models in Section 3.4 and discussing the model core's expressive power together with the involved modeling effort (Section 3.5). Finally, Section 3.6 differentiates the model core from related work that also focuses on security models to express a variety of models.

3.1 Security Model Family Tree

In analogy to a family tree in genealogy describing relations between individuals, a security model family tree describes relations between security models. In contrast to a genetic family tree, a security model family tree does not represent the parental lineage of individuals.

Instead, it represents relations between models based on shared model abstractions like attributes or types. Security models descend from each other whenever they inherit at least one model abstraction to specify specific model components. In this way, a security model family tree enables us to identify the common model core.

In order to compile a security model family tree, we refer to the existing levels of abstraction between models (Section 2.3). More precisely, we discuss how these abstractions represent lineage relations between models and form a model family tree. We first propose a family tree for each of the considered model domains exposing shared model components within these domains. Second, by merging these trees we form a single family tree and thus show that models share components – the model core – even beyond domain borders.

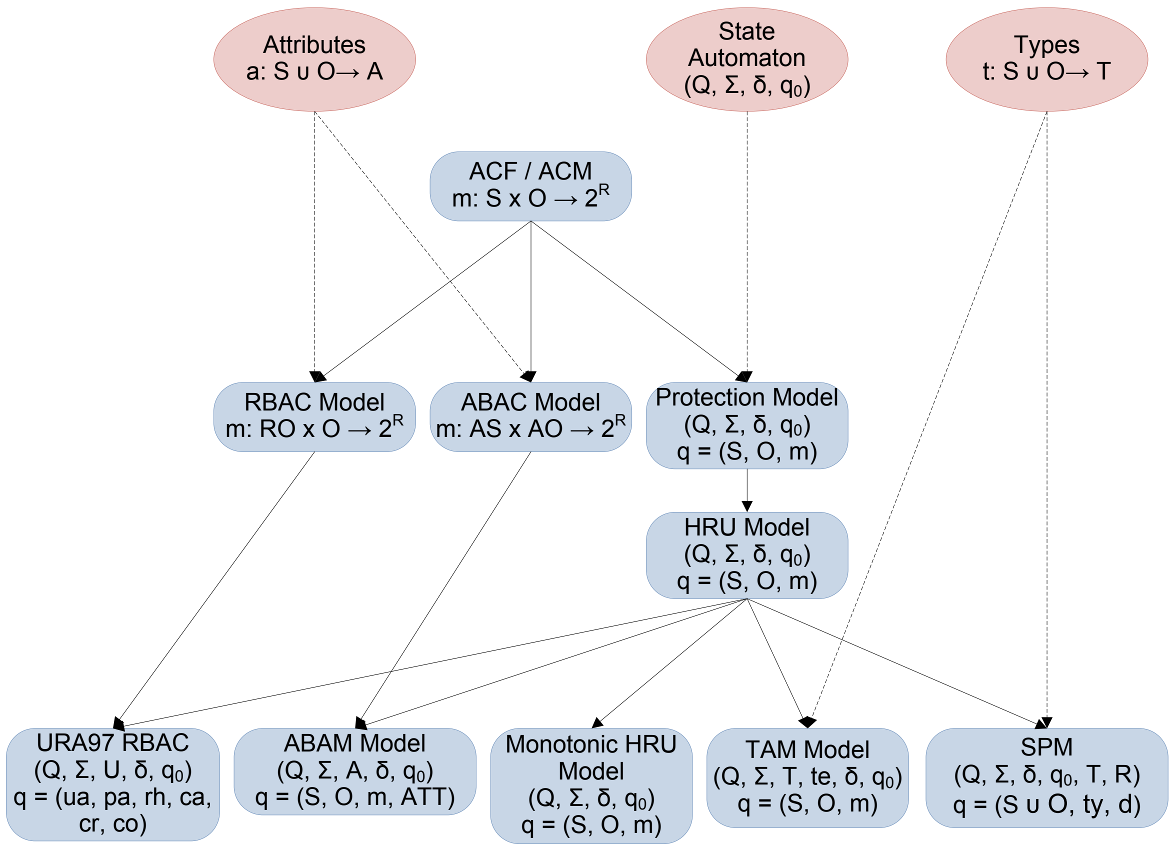

Figure 3.1: A Family Tree of Access Control Models

Comparing components of access control models (Figure 3.1), it becomes apparent that any model contains components to assign permissions to subjects and objects. Lampson's ACF [145] or rather the isomorphic ACM defined as $acm : subjects \times objects \to 2^{rights}$ directly assigns permissions to subjects and objects and is thus on a rather elementary abstraction level. RBAC and ABAC models introduce a new level of abstraction that reflects the models' application-oriented level of abstraction by assigning permissions to subjects via roles or attributes. This is usually modeled by an ACM, e.g., $acm : roles \times objects \to 2^{rights}$, and additional components that for instance map roles to subjects. Hence, all these access control models share the concept of ACMs to model permission assignments and on this account we consider Lampson's ACM model to be the ancestor of the access control model family from which any model emerges directly or indirectly.

Similarly, all access control models supporting policy dynamics are based on a deterministic automaton (Q, Σ, δ, q_0). Forerunner of these dynamic models is the protection model of [69, 101] where each $q \in Q$ is Lampson's ACM. By introducing new levels of abstractions, dynamic models emerge from [69,101] and its more formalized descendant – the HRU calculus [107,108] – inheriting the automaton with its ACM-based state space and adapting it to reflect the new models' levels of abstraction. For example, the ABAM model [254] combines the ACM with subject and object attributes; and both the ACM as well as the attribute assignments are state components. Hence, all dynamic access control models share a deterministic state automaton to describe policy dynamics. For all holds that a state of the automaton always represents a policy's protection state, e.g., a system's right, role, or attribute assignments, and policy dynamics is represented by changing right, role, or attribute assignments via state transitions.

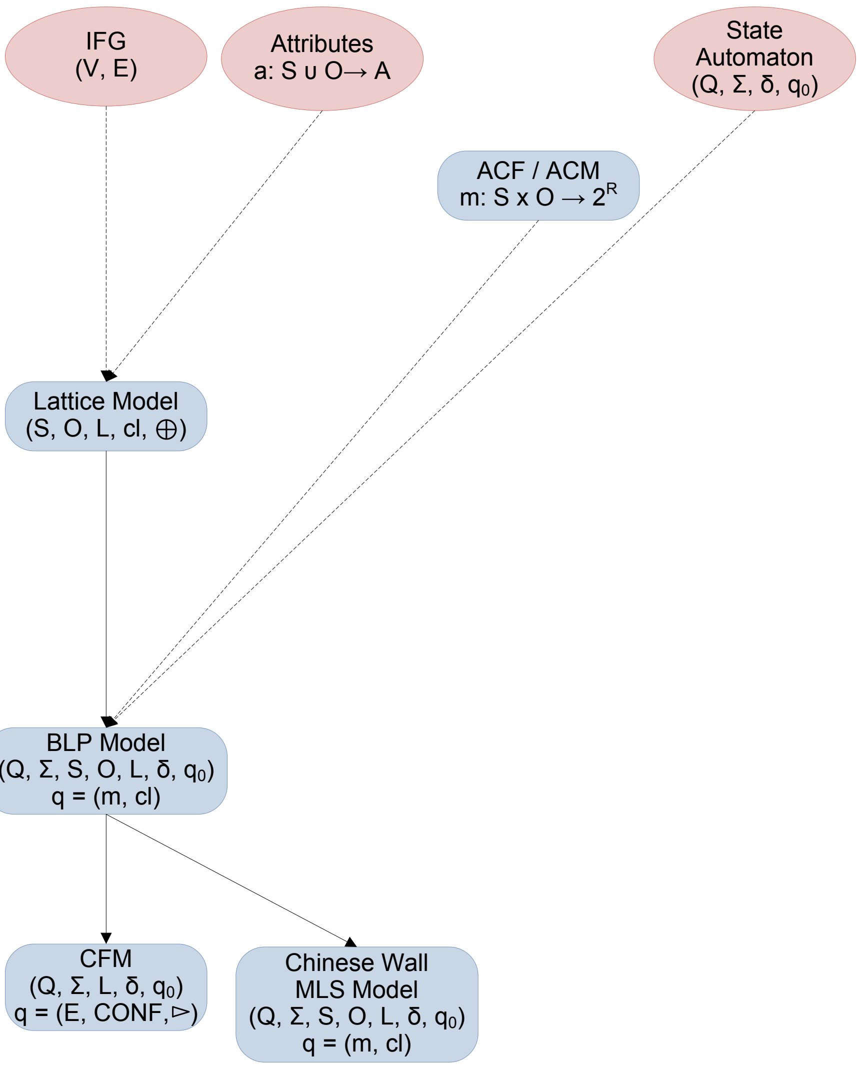

Figure 3.2: A Family Tree of Information Flow Models

In analogy to access control models, we now sketch a family tree of information flow models (Figure 3.2) by comparing their components. Common to all information flow models is a lattice representation $L = (SC, \leqslant)$ of an information flow graph (IFG) and a subject/object attribute assignment (classification function) $cl : S \cup O \rightarrow SC$ that assigns a security class to subjects and objects. Therefore, we consider the lattice model of Denning [67] to be the ancestor of the information flow model family.

On top of that, many of the discussed dynamic information flow models apply an deterministic automaton (Q, Σ, δ, q_0) to model policy dynamics. Forerunner of dynamic information flow models is the BLP model [26, 27]; and by introducing new levels of abstraction, dynamic models emerge from the BLP model. Models inherit the automaton and the lattice, and adapt them to reflect the models' levels of abstraction. For example, CFM [92] inherits the HRU derivative from BLP for modeling policy dynamics and implementing the lattice but adds a confine function to its state space. For all models it is true that a state of the automaton always represents a policy's protection state, e.g., a system's subject/object classification into security levels, and policy dynamics is modeled by a state transition function that for example implements reclassification rules.

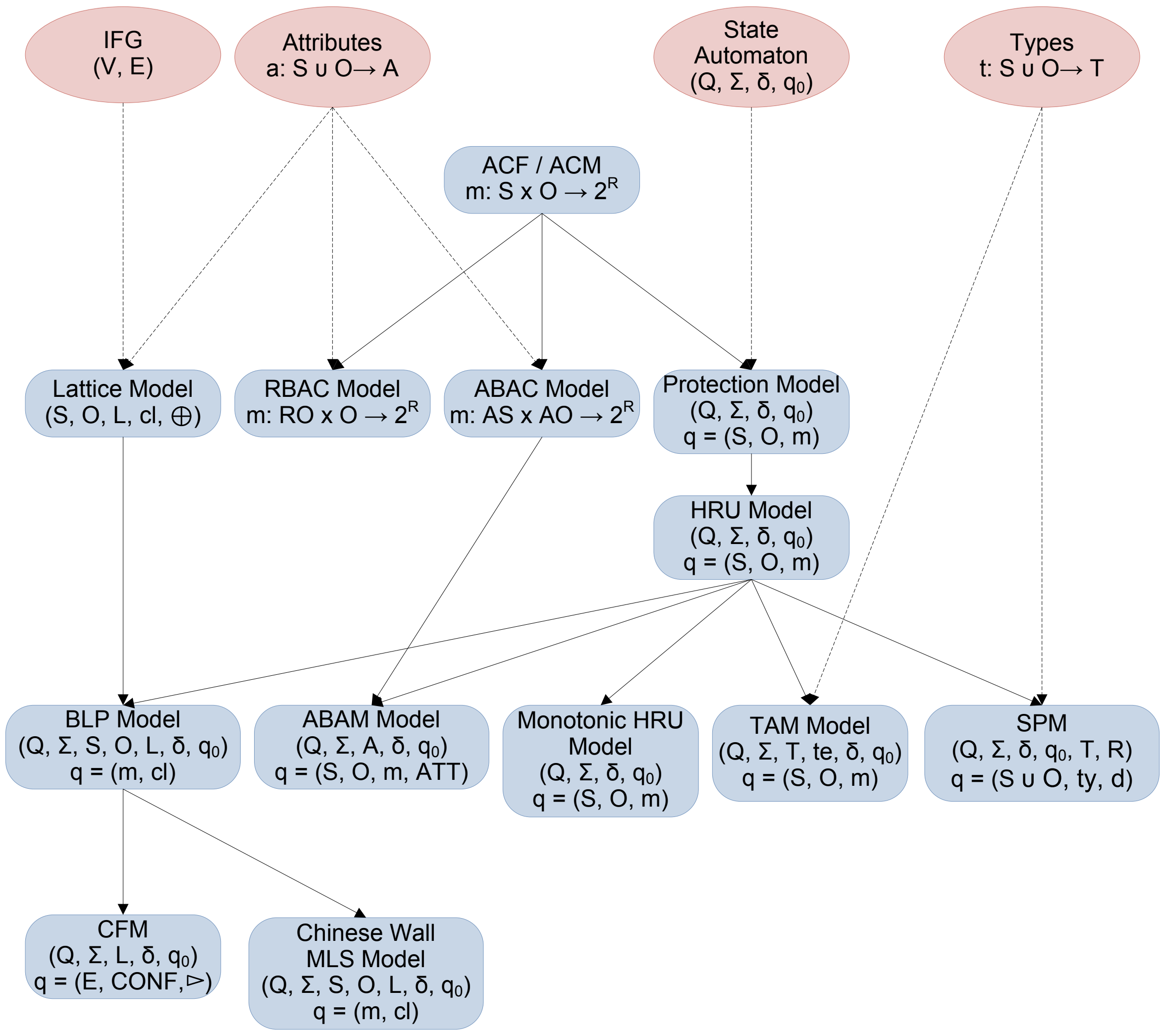

Figure 3.3: A Family Tree of Security Models

Figures 3.1 and 3.2 reveal that security models do not only share domain-specific components but also a domain-independent abstraction. This sets the course for merging both trees into a single family tree (Figure 3.3), which then contains lineage relations beyond domain borders. More precisely, all dynamic access control and information flow models share the following domain-independent abstractions:

1. A deterministic automaton to model policy dynamics.
2. A state transition function that describes the dynamic behavior of a policy, e.g., by changing right, role, or attribute assignments.
3. An automaton state that represents a policy's protection state, e.g., its right, role, or attribute assignments, depending on a model's domain.
4. Static components that are not modified during policy runtime. Just like states, static components depend on the model domain.

In the remainder of this dissertation we refer to the automaton as the *security model core*. Its formal definition is provided in the following section.

3.2 Core Definition

The proposed core-based model engineering approach (Section 3.3) along with the proposed approach for engineering causal TCBs (Section 4) requires a sound formal basis. Therefore, this section provides the formal definition of the identified common model core and thus establishes a foundation for the following sections.

The model core is a deterministic automaton. It generalizes the HRU automaton [141] and then allows for its specialization. Security models can be derived from the model core by core specialization inspired by object-oriented design.

Definition 3.1 (Security Model Core) The *security model core* is a tuple $(Q, \Sigma, \delta, q_0, E)$ combining a deterministic automaton (Q, Σ, δ, q_0) with a vector E of static extensions where Q is a set of protection states, Σ is a finite set of inputs, $\delta : Q \times \Sigma \to Q$ is the state transition function, and $q_0 \in Q$ is the initial protection state.

The **state set** $\mathbf{Q} = \times_i^n D_i, n \geqslant 1$, represents the set of protection states of a security model and is in general infinite[1]. A protection state $q = (d_{1_q}, ..., d_{n_q})$ may consist of several different components $d_i \in D_i$ where each d_i is a set, relation, or function with its individual meaning, e.g., subject set, role set, or role assignment function. Thus, a protection state is an n-tuple consisting of n structurally different components. These components are dynamic and can be modified by policy-specific inputs.

Besides a description of the dynamic model state, the definition also contains static model components. The **extension vector E** represents a virtual type for such static model components $E = (e_1, e_2, ..., e_h), h \geqslant 0$, where each extension e_i is a set, relation, or function with its model-specific interpretation, e.g., type set, security level set, or a dominance relation of security levels. Thus, the extension vector contains h structurally different components.

The **input set** $\mathbf{\Sigma} = C \times X$ is defined by the finite set C of policy-specific commands and an input vector set X of command parameters. Policy-specific commands are those commands whose executions are protected by a policy and which modify protection states, e.g., *adduser*

[1]Note that a security model can be engineered in such a way that its set of protection states is finite. The model core is then specialized to a deterministic finite automaton (DFA).

or *chmod* in a Unix/Linux-based system. Command-specific input parameters are modeled by l-dimensional vectors x so that for any input holds: $x \in X$ with $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$ where any vector element of x may have an individual type. This is modeled by a type set $T = \{D_1, ..., D_n, d_{1_q}, ..., d_{n_q}, e_1, ..., e_h, 2^{e_1}, ..., 2^{e_h}\}$, which is a set of all components of a model's state set, state, and extension vector, along with the power sets of the extension vector components[2].

Model dynamics are specified by the **state transition function $\boldsymbol{\delta}$** by executing a policy-specific command $c \in C$. Any command defines authorization rules that consist of (i) conditions to check whether executing the command is allowed and (ii) primitive actions to modify the model's state. Thus, each command c is a tuple $(x_c, \mathit{Cond}_c, \mathit{Prim}_c)$ where x_c is the vector of formal parameters (represented by a vector of literals), Cond_c is a set of conditions and Prim_c is a sequence of primitive actions. Executing a command c depends on the current model state and its input parameters x_p and may result in modifying the state so that the automaton enters a subsequent state $q' \in Q$ where $\delta(q, (c, x_p)) \mapsto q'$ and x_p is some value of x_c. More precisely:

Definition 3.2 (State Transition Function) The *state transition function* $\delta : Q \times \Sigma \rightarrow Q$ is a function that transitions a model's protection state q to a subsequent state q' iff all of the required conditions are met.

$$\delta(q, (c, x_p)) \mapsto \begin{cases} q', & \forall \mathit{cond} \in \mathit{Cond}_c : \mathit{cond}(q, x_p) = \mathit{true} \\ q, & \text{otherwise} \end{cases}$$

$q \in Q$ is the current model state, $c \in C$ is a policy-specific command, $x_p \in X$ are the input parameters according to the formal parameter vector x_c, and $\mathit{cond} \in \mathit{Cond}_c$ are conditions specific to c.

The subsequent state is computed as

$$q' = \mathit{prim}_k(\mathit{prim}_{k-1}(...(\mathit{prim}_2(\mathit{prim}_1(q, x_p), x_p), ...), x_p), x_p)$$

where $\mathit{Prim}_c = (\mathit{prim}_1, \ldots, \mathit{prim}_k)$ is the sequence of primitive actions specific to c, $k > 0$. We refer to *PRIM* as the set of all primitive actions of a model. Accordingly, the set *COND* contains all possible conditions of a model.

Definition 3.3 (Condition) A *condition cond* : $Q \times X \rightarrow \mathit{Bool}$ is a boolean function based on first-order logic consisting of one or more clauses for expressing requirements that need to be met to enable primitive actions to modify the model state.

$$\mathit{cond}(q, x_p) \mapsto \begin{cases} \mathit{true}, & \mathit{cl}_1(x_p) \wedge \ldots \wedge \mathit{cl}_m(x_p) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$q \in Q$ denotes the model state, $x_p \in X$ are the input parameters according to the formal parameter vector x_c, and $\mathit{cl}_1 \wedge \ldots \wedge \mathit{cl}_m$ are conjunctively combined clauses specific to *cond,* $m > 0$.

The main purpose of primitive actions is to define rules which allow for creating any protection state that is needed by a security model. These rules then describe, how the states are created by modifying the state components $d_{1_q}, ..., d_{n_q}$.

[2]Note that a function must be denoted as left-total and right-unique relation so that its power set can be created.

Definition 3.4 (Primitive Action) A *primitive action* $prim : Q \times X \to Q$ modifies a model's protection state depending on the model state and the vector of input parameters: $prim(q, x_p) \mapsto q'$. $q \in Q$ is the current policy state, $x_p \in X$ are the input parameters according to the formal parameter vector x_c, and $q' \in Q$ is the subsequent state.

Summary By generalizing the HRU automaton and providing a formal definition, the foundation for specializing the model core has been laid. Core components that can now be specialized to a model are

(i) the protection state $q = (d_{1_q}, ..., d_{n_q})$, and thus the state set $Q = \times_i^n D_i, n \geqslant 1$,

(ii) the extension vector $E = (e_1, e_2, ..., e_h), h \geqslant 0$,

(iii) the finite primitive set $PRIM = \{prim_1, \ldots, prim_l\}$, $l \geqslant 1$,

(iv) the finite condition set $COND = \{cond_1, \ldots, cond_k\}$, $k \geqslant 0$, and

(v) the input vector set X of the input set Σ.

We discuss core specialization methods in the following section.

3.3 Core Specialization

Specializing the common model core allows for tailoring the core components to meet the needs of a model's individual characteristics. The notion of specializing the model core goes back to generalization/specialization concepts of object-oriented (OO) software design. In OO design, a subclass is derived from a super class by inheriting attributes, operations, and constraints of the super class with the goal of reusing software. Thus, subclasses are allowed to add new features, e.g., new attributes and operations, or to specialize inherited operations in order to express individual class characteristics. In the context of security models, the model core corresponds to a super class and any security model corresponds to a subclass that can be derived from the model core. Consequently, models inherit the core components Q, Σ, δ, q_0, and E, which may then be specialized.

The result of core specialization is a model-specific state automaton with a specialized model state, a specialized state transition function, as well as a specialized extension vector. The model core is specialized following six steps; details on each of the six steps are given below. On top of that, each step is demonstrated by means of an example based on the HRU model. We have chosen the HRU model here, because it is well-known and allows the reader to focus on the steps of core-based engineering. The latter is also supported because both, the model core and the HRU model, apply the same levels of abstraction. Besides that, Section 3.4 provides additional examples in terms of more previous and more application-oriented models.

1. Model Components: identifying the individual components of a model.

2. State Set: combining dynamic model components to specialize the state set.

3. Extension Vector: combining static model components to specialize the extension vector.

4. Input Vector Set: deriving the input vector set.

5. Primitive Actions: defining primitive actions that modify the dynamic model state.
6. Interrelations: identifying interrelations between dynamic and static model components and hence the set of conditions.

1. Model Components: It is necessary to identify all components of a model first. In this context, some typical model components are: (i) entities that are protected by a policy, including passive entities (such as files or directories) and active entities (e.g., users and processes) as well as the entities' actions, (ii) entity attributes such as types, rights, or roles, and (iii) relations between the identified model components, e.g., assignment attributes to entities. Once all components are identified, their types have to be defined. Common types are set, function, relation, tuple, and any combination of these.
Result is a finite set of model components $M = \{m_1, m_2, ..., m_n\}$, $n \in \mathbb{N}, n \geqslant 1$, which are described by a name and a type.
Example The HRU calculus defines an infinite subject set S, an infinite object set O, a finite set of access rights R, and a function acm that maps a pair of subjects and objects to a subset of R. Thus, $M = \{S, O, R, acm\}$.

2. State Set: Now, the state set can be composed by selecting all those model components contained in M that may be modified during policy runtime. In terms of object orientation, $Q = \times_i^n D_i, n \geqslant 1$, represents a virtual type for the dynamic model component. D_i are sets of dynamic state components d_i, which represent those model components that can be modified by primitive actions.
Result is an n-tuple $q = (d_{1_q}, \ldots, d_{n_q})$, where $d_{i_q} \in D_i$ and $d_{i_q} \in M$. q_0 must then be initialized according to the state space $Q = \times_i^n D_i, n \geqslant 1$, where $q_0 = (d_{1_0}, ..., d_{n_0})$.
Example Dynamic components of the HRU model are the subject and object sets along with the matrix; any state $q = (S_q, O_q, acm_q)$ then contains the state-specific subject set S_q, the state-specific object set O_q, and a state-specific acm_q. Thus, $Q = 2^S \times 2^O \times ACM$ where $S_q \subseteq S, O_q \subseteq O, acm_q \in ACM$, and $ACM = \{acm | acm : S \times O \rightarrow 2^R\}$.

3. Extension Vector: After having selected all state components from M in step 2, M purely consists of static components that are not part of the model's state and thus are not modifiable by primitive actions. These components can now be inserted in the extension vector of the model core.
Result is a model-specific extension vector $E = (e_1, \ldots, e_h)$ with h static model components, $h \geqslant 0$, $e_i \in M \backslash \{d_j\}, 1 \leqslant i \leqslant h$, and d_j is a tuple element of q.
Example The right set of the HRU model cannot be modified by δ. For this reason, the right set R is considered static. Thus, there is one element R in the extension vector that is added to the model core $(Q, \Sigma, \delta, q_0, (R))$.

4. Input Vector Set: Having specialized a model's state components and static extensions, the input vector set X based on the set of input types T can be derived.
Result is an l-dimensional input vector set $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$ where the type set T consists of all the components of Q, q, E, and the power sets of E's components. Thus, $T = \{D_1, ..., D_n, d_{1_q}, ..., d_{n_q}, e_1, ..., e_h, 2^{e_1}, ..., 2^{e_h}\}$ where $Q = \times_i^n D_i$, $d_{i_q} \in D_i$ and e_j is an element of E where $1 \leqslant j \leqslant h$.

Example Based on the steps before, the type set T of the HRU model is defined as $T = \{2^S, 2^O, ACM, S, O, acm, R, 2^R\}$.

5. Primitive Actions: Primitive actions are atomic operations modifying the state components of the model. Policy semantics confine the set of all possible functions to those that are actually needed. For example, based on real-world IBAC policies, where the ability to create or remove privileges is an important requirement, models usually contain primitive actions that add elements to a set or relation, or delete elements. Depending on the results of the previous step, a set of primitive actions can now be defined that allows for creating any protection state that is needed by a policy.
Result is a set $PRIM = \{prim_1, \ldots, prim_l\}$, $l \geqslant 1$, where every primitive action $prim_i \in PRIM, 1 \leqslant i \leqslant l$, is a function $prim_i : Q \times X \to Q$ where $prim_i(q, x_{pl}) \mapsto q'$ with $q' \in Q$.
Example The HRU calculus defines six primitives: S_q and O_q are modified by creating and destroying subjects/objects, and acm_q is modified by entering and deleting rights. For example, the *enter*-primitive enters a right $r \in R$ into an ACM cell $m(s, o)$. This is rewritten in core notation as $prim_{enter\_r}(q, (x_s, x_o)) \mapsto (S_q, O_q, m_q(x_s, x_o) \cup \{r\})$.

6. Interrelations between Dynamic and Static Model Components: In order to relate extensions to the policy state, security models define interrelations between dynamic and static model components by means of using the conditions of the transition function. Thus, δ makes policy decisions not only based on state components but also on core extensions.
Result is a set of conditions $COND = \{cond_1, \ldots, cond_k\}, k \geqslant 0$, that relates the extension vector E to the dynamic model component q. Each condition $cond \in COND$ is a function $cond : Q \times X \to Bool$ where $cond(q, x_p) \mapsto \{true, false\}$.
Example The HRU calculus defines a Boolean expression based on one clause $r \in m(x_s, x_o)$ that checks whether a subject x_s has the right r on an object x_o by verifying whether r is element of cell $m(x_s, x_o)$. In core notation the HRU condition is defined as $cond_{HRU} : Q \times X \to Bool$ where

$$cond_{HRU}(q, (x_s, x_o)) \mapsto \begin{cases} true, & r \in m_q(x_s, x_o) \\ false, & \text{otherwise} \end{cases}$$

Having employed all specialization options, the result of core-based model engineering is a core-based model representing the individual characteristics of a specific model. This model can now be applied to model application-specific security models as shown in Appendix B.

Summary This section has proposed a core-based model engineering method for access control and information flow models. Inspired by object-oriented software design, where specialized subclasses are derived from a super class, the common model core corresponds to a super class and models (corresponding to subclasses) are engineered by specializing the common model core. The outcome are core-based models with shared model abstractions.

The contributions of core-based model engineering can be exploited in four ways: (i) reuse of core components, (ii) model-independent security analysis and implementation methods, and (iii) model-independent tool support for model engineering, analysis, and implementation, and (iv) development of causal TCBs that are tailored to specific security models. In order that these merits have a precise foundation, we have provided a formal definition of the core along with an engineering method.

3.4 Model Re-engineering

After having discussed how to tailor the common model core to individual models, the goal of this section is to show the generality and feasibility of core-based model engineering by means of three models: an MLS, an RBAC, and an ABAC model. Even though these models belong to different generations and were developed for different application scenarios, we can demonstrate that they share common abstractions. Indeed, we show that the MLS model contains the model core although it is not obvious. In contrast, the RBAC and the ABAC model are static and do not contain the model core by any means. However, we show that their design principles enable core-based model engineering, which results in a dynamic RBAC and a dynamic ABAC model. By re-engineering all three models, the model core becomes obvious, which then allows for applying model-independent methods and tools. Moreover, once we have applied these model-independent engineering methods, the models can be executed in a model-independent runtime environment for core-based models (Section 4).

Therefore, this section briefly sketches the three models (Sections 3.4.1, 3.4.3, and 3.4.6) and introduces an RBAC policy that provides the basis for re-engineering the RBAC model (Section 3.4.4). On top of that, the RBAC policy is used as running example in the following sections. A discussion on the re-engineering process for each model follows in Sections 3.4.2, 3.4.5, and 3.4.7. Note that the resulting core-based models are too extensive to discuss all the details in this section. Hence, we focus on excerpts of each of the six model engineering steps; the complete models are illustrated in Appendix A.

3.4.1 Multilevel Security Models

For more than 30 years, multilevel security models such as [26,35,92] have been an important utility for modeling information flow security policies. The main application area of MLS policies have been highly trustworthy IT systems in military and governmental environments. Still, the number of IT Systems enforcing MLS policies is steadily increasing, e.g., trusted operating systems such as SELinux [177,223,224], Security Enhanced BSD [240,246,247], or Oracle Solaris 10 [186], database management systems such as the Oracle Database 11g Release 1 [185], and hypervisors such as XenClient XT by Citrix [54]. One important application scenario is dynamic web service composition, where complex services available on the Web are composed at design time by service composition planners such as SHOP2 [118]. Hutter et al. [118–120] have developed an MLS model and a system implementing this model for automatically composing web services for collaborative business processes and health care composition plans.

The wide variety of contemporary IT systems and application scenarios shows that MLS models are by no means deprecated. Inspired by the BLP model [26, 27], many new models and IT systems have emerged adapting the basic concepts towards modern application scenarios. Thus, even though MLS models were developed 40 years ago, they still represent a significant class of models. For this reason, we demonstrate core-based model engineering by means of an contemporary MLS model – the model for dynamic web service composition [118–120]. Since it was derived from the BLP model, we briefly sketch the main concepts of the BLP model in more detail before introducing the model of Hutter et al. [119,120].

The BLP model $(Q, \Sigma, \delta, q_0, (L, S, O, R))$ combines a lattice with components of the HRU model. The lattice $L = (C, \leqslant)$ represents a hierarchy of a finite set of security classes C modeled by the dominance relation $\leqslant$ as a total ordering. Every state $q \in Q$ of the automaton is a tuple $q = (m_q, cl_q)$, where m_q is an ACM as defined in Section 2.3.2 and

$cl_q : S \cup O \rightarrow C$ is a classification function assigning a security class to each subject and object. In order to modify m_q and cl_q, the model defines four primitive operations that pairwise perform modifications on the matrix and the classification function: *enter right* into and *delete right* from the matrix, *classify* and *reclassify* subjects/objects. Entering a right into the matrix is controlled by the classification function and the lattice. This is established by the *Simple-Security Rule* and the $\star$*−Property*, which map legal information flow onto specific rights (*read* or *write*) in the matrix. Thus, the matrix may only contain those rights that are allowed by the lattice and the classification function. Since the matrix does not need to be equivalent to the lattice, information flow can be controlled by two equivalent conditions: (i) $\exists r \in R : r \in m_q(s, o)$ checking whether the corresponding right (*read* or *write*) is in a matrix cell, or (ii) $cl(o) \leqslant cl(s)$ that checks whether the security class of s dominates the security class of o in order to allow that information flows from o to s (respectively, $cl(s) \leqslant cl(o)$ allowing information to flow from s to o).

The BLP model has inspired numerous models such as the semi-formal model for web service composition policies [119, 120]. In analogy to the BLP model, the web service model consists of a set of objects O, a set of subjects WS called web services, a classification function $cl : WS \cup O \rightarrow C$, and a lattice $L = (C, \leqslant)$ to model a security class hierarchy. However, the definition of the lattice differs. Similar to the compartments of the BLP model, [119,120] introduce different types of information. Instead of assigning compartments to objects/subjects, the security level of a web service is subdivided with respect to the type of information. Thus, different lattices are combined, i.e., each lattice $(C_1, \leqslant_1), (C_2, \leqslant_2), \ldots, (C_n, \leqslant_n)$ represents one type of information, to a *composed* lattice $L = (C, \leqslant)$ by $C = C_1 \times C_2 \times \ldots \times C_n$ and $(c_1, c_2, \ldots, c_n) \leqslant (c'_1, c'_2, \ldots, c'_n)$ iff $c_i \leqslant c'_i$ holds for all $1 \leqslant i \leqslant n$. The number of lattices n depends on the policy the model is engineered for, e.g., the model defines two types of information for a travel agency policy [119]: *location* and *finance* resulting in a composed lattice $L = (LO \times FI, \leqslant)$. In contrast to the BLP model, this model does not contain an ACM and models policy dynamics semi-formally.

Dynamically composing web services in order to fulfill a specific task results in the problem that a customer usually does not know all web services that will be involved [119]. Consequently, whenever a new web service is added to the composition plan, it needs to be classified. For this purpose, the model contains a delegation function $del : WS \cup O \rightarrow C$ that allows a web service to classify a new web service according to the *delegation classification*. $del(ws) \mapsto (c_1, ..., c_n)$ denotes the maximal classification a web service may delegate to an unknown web service [119].

Having introduced all concepts of the semi-formal model of Hutter et. al., the following section discusses how to re-engineer this model.

3.4.2 MLS Model for a Web Service Composition System Policy

The model of Hutter et al. [119, 120] considers policy dynamics in a semi-formal way and hence contains the model core even though it is not obvious. The challenge of model engineering (Section 3.3) is now to expose the model core by defining the state set and primitive operations, and relating state components and static extensions. In doing so, we follow the notions of the BLP model, because of the similarities of both models.

Step 1 now defines all model components. The model components are analogous to the model components of Hutter et. al. and thus M consists of seven elements $M = \{WS, O, cl, del, LO, FI, \leqslant\}$.

Afterwards, step 2 determines the state components[3]. Since the set of web services can be modified by adding new web services and hence the classification function and the delegation function are modified by adding mappings for new web services, WS, cl, and del are considered dynamic. Consequently, the state set is defined as $Q = 2^{WS} \times CL \times DEL$ where $CL = \{cl \mid cl : WS \cup O \to LO \times FI\}$ and $DEL = \{del \mid del : WS \cup O \to LO \times FI\}$. Any state is a tuple $q = (WS_q, cl_q, del_q)$ where $WS_q \subseteq WS$, $cl \in CL$, and $del \in DEL$.

Step 3 specializes the extension vector by including the remaining components O, LO, FI, and $\leqslant$ in the extension vector E, resulting in a model $(Q, \Sigma, \delta, q_0, (LO, FI, \leqslant, O))$.

After having specialized the model's dynamic and static components, we can derive the type set T of the input vector set X in step 4. According to the specialized core components Q, q, and E, $T = \{2^{WS}, CL, DEL, WS, cl, del, LO, FI, \leqslant, O, 2^{LO}, 2^{FI}, 2^{\leqslant}, 2^{O}\}$.

To precisely specify the legal modifications of a model state, step 5 defines the set of primitive actions. Since web services can only be added and not deleted, and delegations and classifications can only be granted once for the new web services, primitives for deleting web services and for reclassifying web services and objects are not required. Thus, we define a primitive for adding web services and a primitive for classifying web services. The entire set of primitives is given in Appendix A.2. Note that in order to define the primitives, a binary operation called *functional overriding* $\oplus$ is required; it is defined as follows. Given two functions $f : X \to Z$ and $g : Y \to Z$; the overriding of f by g is defined as $f \oplus g : (X \cup Y) \to Z$ where $f \oplus g\,(x) \mapsto \begin{cases} g(x), & x \in Y \\ f(x), & \text{otherwise.} \end{cases}$

$$addWebService : Q \times X \to Q$$
$$addWebservice(q, \{x_{ws_1}, \ldots, x_{ws_n}\}) \mapsto (WS_q \cup \{x_{ws_1}, \ldots, x_{ws_n}\}, cl_q, del_q)$$

$$classify : Q \times X \to Q$$
$$classify(q, (x_{ws}, x_l, x_f)) \mapsto (WS_q, cl_q \oplus \{(x_{ws}, (x_l, x_f))\}, del_q)$$

Finally, we need to relate the extensions to the model state. In doing so (step 6), we follow the notation of MLS models by rewriting their conditions, e.g.,

$$cond_{MLS_1} : Q \times X \to BOOL$$
$$cond_{MLS_1}(q, x_{ws}, x_o) \mapsto \begin{cases} true, & clause_{MLS_1}(q, cl_q(x_o), cl_q(x_{ws})) \\ false, & \text{otherwise} \end{cases}$$

$$clause_{MLS_1} : Q \times X \to BOOL$$
$$clause_{MLS_1}(q, x_l, x_f) \mapsto \begin{cases} true, & x_l \leqslant x_f \\ false, & \text{otherwise} \end{cases}$$

Moreover, the model requires a condition that checks whether copying the delegation classification of one web service to a new one is allowed. Thus, the core-based MLS model contains a total of three conditions including three clauses.

Summary We have demonstrated core-based model engineering by means of an MLS model for web service composition systems [119, 120]. This model is an offspring of the BLP model [26, 27] that has long been proven relevant in the security community.

[3]Note that the authors of this model have given some freedom in the interpretation of which model components may be modified and how to do so. Due to demonstration purposes we have restricted the use case policy in such a way that the resulting model is straightforward and comprehensible.

The MLS model already contains the model core (even though it is not obvious) by the model components *WS*, *cl*, and *del*, since they may change during policy runtime. However, the original model does not formally define how to do so. Hence, the challenge of re-engineering has been to define the primitive actions that modify the model state. The result is a dynamic MLS model with an exposed model core, which is now unlocked to a family of engineering methods and tools.

3.4.3 Role-based Access Control Models

In the early 1990s, commercial applications required the ability to specify and enforce enterprise-specific access control policies in order to streamline the typically burdensome process of authorization management [86]. This resulted in the development of role-based access control models, e.g., [86,87,184,205,210,212,229], which have quickly become relevant for expressing and implementing application-oriented access control policies. This is reflected by a wide variety of commercial and noncommercial applications, e.g., [45,83,187,232], which are widely scattered in the sectors of government and military, bank and finance [213], insurance and health care [77,221]. More case studies are discussed [86,177].

Among the wide variety of RBAC models, Sandhu's RBAC model family [212] is the basis for the most relevant models: RBAC_0, RBAC_1, RBAC_2, and RBAC_3. These models are standardized by the American National Standard 359-2004 [88,152,205] and many researchers and software vendors have provided methods, e.g., for role engineering [179], and tool support for these models [86].

Unfortunately, these RBAC models have a severe drawback. They focus on specifying static snapshots of RBAC policies without considering modifications during their runtime. For example, commercial IT systems such as [187] show that defining new roles and assigning them to users are indispensable features of RBAC systems. Moreover, numerous publications, e.g., [157,228,229], have illustrated that modeling a static snapshot of a policy is not sufficient for analyzing RBAC models with respect to their safety properties (HRU models in Section 2.3.2). The promising potential of RBAC models is yet to be fully exploited. On this account, literature proposes to upgrade existing RBAC models by model dynamics [157,228,229].

Our approach is to structurally engineer a dynamic RBAC model based on Sandhu's static model family. The goal is to demonstrate that RBAC models, even though they are originally static, share the common model core. In doing so, we refer to the RBAC_3 model and engineer it to be suitable for a Health Information System (HIS) policy that is used as running example in this dissertation. The re-engineered RBAC model then enables to formally describe the sample policy, which we then use for demonstrating causal TCB engineering. On this account, this section gives a short overview of RBAC models; a comprehensive discussion can be found in [86,205,212]. Afterwards Section 3.4.4 introduces the HIS policy, which provides the basis for re-engineering the Sandhu's RBAC_3 model in Section 3.4.5.

Figure 3.4 illustrates the components of Sandhu's RBAC_3 model. Basically, any RBAC model consists of four sets: users U, roles R, permissions P, and sessions S (also known as subjects). Users assume roles, modeled by a user-to-role assignment relation $UA \subseteq U \times R$, and roles are associated with permissions by a role-permission assignment relation $PA \subseteq P \times R$. The set of permissions $P = 2^{(OP \times O)}$ is defined by relating a policy's operations and objects; a permission is a set of tuples where each tuple consists of an operation that is allowed to be executed on an object.

A session represents a user in the system and carries out all requests by the user. This is

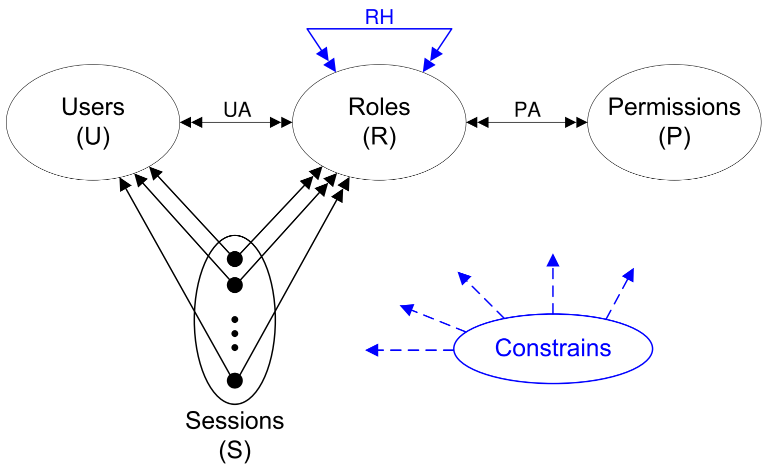

Figure 3.4: RBAC Security Model [212]

modeled by a mapping $user : S \to U$ that maps a session to a user. Note that a user may have more than one session, e.g., he may be logged in to a system multiple times. However, any session strictly represents only one user.

A session activates a subset of the user's roles by a function $roles : S \to 2^R$ where $roles(s) \mapsto \{r \in R \mid (user(s), r) \in UA\}$. This means, a session can only activate roles that are assumed by its user. Once a session has activated some of the user's roles, it is authorized to perform an operation on an object if it has an activated role with a permission that allows the operation to be executed on the object. This is modeled by a function $access : S \times OP \times O \to Bool$ [86] where

$$access(s, op, o) \mapsto \begin{cases} true, & \exists r \in R, p \in P : r \in roles(s) \wedge (p, r) \in PA \wedge (o, op) \in p \\ false, & \text{otherwise} \end{cases}$$

So far, these definitions are common to all RBAC models. Unique features of the RBAC$_3$ model are role hierarchies and separation of duty constraints that can be defined on any model component. We motivate and illustrate these concepts in the following.

Role Hierarchies are a natural means for structuring roles in order to reflect an organization's authority or responsibility lines [212]. It is common practice that roles within an organization overlap and thus share some of their permissions. This results in permission redundancies and therefore also results in increased maintenance effort. To avoid redundancy Sandhu [212] introduced role hierarchies for permission inheritance between roles. For example, when a role *doctor* inherits permissions from a role *nurse*, all permissions of the *nurse* are added to the individual permissions of the *doctor*. In other words, the nurse's permissions become a subset of the doctor's permissions.

Role hierarchies are usually modeled by a partial order $RH \subseteq R \times R$ (denoted by $r \succeq r'$) where $\forall r, r' \in R : r \succeq r' \Leftrightarrow$ all permissions of r' are also permissions of r and all users of r are also users of r' [86]. This leads to a redefinition of *access* in order to perform access control checks in the presence of role hierarchies [86]. Access is thus only granted when the user's session activates a role inheriting a permission from another role that allows an operation to be executed on an object. Since role hierarchies are reflexive, anti-symmetric, and transitive, *access* does not exclude the roles (and their permission) directly activated by the user's session:

$$access(s, op, o) \mapsto \begin{cases} true, & \exists r, r' \in R, p \in P : r \geq r' \wedge r \in roles(s) \wedge \ (p, r) \in PA \\ & \wedge (o, op) \in p \\ false, & \text{otherwise} \end{cases}$$

It is also common practice that individuals within an organization may not assume certain duties at the same time – a common principle called separation of duty – in order to constrict the possibility to endanger the organization's business or security. Well known examples are the roles of purchasing manager and accounts payable manager. A user is generally not allowed to assume both duties because this creates the opportunity to committing fraud [212].

Such organizational restrictions are transferred to RBAC models by attaching separation of duty constraints to model components; a prominent representative of such constraints are mutual disjoint roles, where a user may not assume two or more roles at the same time. Depending on whether mutual disjoint roles are defined as static or dynamic separation of duty constraints, i.e., whether they are enforced during authorization-time or runtime (by either constricting UA or $roles$), the modeling of such constraints is manifold, e.g., [139,158]. The health information system policy is based on a simple authorization-time role exclusion with shared privileges. For this reason, the model needs to restrict the user-to-role assignment relation UA. This is typically modeled by a nonreflexive, symmetric, and nontransitive role exclusion relation $RE \subseteq R \times R$ (denoted by $\approx_{ex}$), where $\forall r, r' \in R : r \approx_{ex} r' \Leftrightarrow r$ and r' are mutually exclusive. This means that whenever a user tries to assume a new role, the model has to ensure that the new role and any other role already assumed by the user are not mutually exclusive.

Based on these concepts, the RBAC$_3$ model is able to specify static snapshots of RBAC policies, including organizational role hierarchies and separation of duty constraints by means of mutually disjoint roles. In the following section, we first introduce the sample HIS policy and afterwards show how the static RBAC$_3$ model is re-engineered to support policy dynamics, in order to formalize the HIS policy.

3.4.4 Use Case RBAC Security Policy

This section briefly introduces an RBAC security policy that is used as a running example in this dissertation. The policy is roughly based on a small real-world HIS for an aged-care facility and was first introduced by [77]. In order to develop a formal model for this policy, [99,228,229] have precisely specified the policy, enhanced it by additional policy rules, and extended the policy with parameters, or some aspects of electronic health records [25].

Due to demonstration purposes, this dissertation uses only a small excerpt of the policy rules based on ten roles, 15 objects, seven operations, a role hierarchy, and separation-of-duty based on role exclusion. Thereby, we adopt the policy limits identified by [229] that lead to a low policy complexity, which suits our demonstration purposes very well: the policy does not support parameters for objects, roles, or permissions, such that it is only suitable for a single department. It focuses on the dynamics of the user-role assignment instead of also considering the dynamics of the role-permission assignment. Hence, the policy's object set, permission set, and role-permission assignment do not change during policy runtime. Besides that, we have added a collection of explicit management functions, which are dedicated to manage the users and their roles, and a role *UserAdmin*, which may perform these functions. For example, the role *UserAdmin* may add new users if the aged-care facility recruits new employees.

Other than that, the policy defines only roles that we consider to be standard for a HIS system, e.g., *Doctor, Nurse, Patient*, and *Manager*. The roles and their permissions are managed in a role hierarchy as shown in Figure 3.5. Here, roles with fewer permissions are located at the bottom such that the role *Employee* has the fewest permission, which are directly inherited by the roles *Receptionist* and *Nurse*.

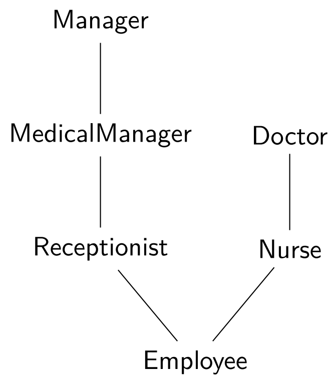

Figure 3.5: Role Hierarchy of RBAC HIS Policy [178]

Additionally, roles are mutually exclusive based on the user-role assignment. For example, a user who is assigned the role *Doctor* may not have the roles *Receptionist* and *Manager*.

Objects of the policy are also standard objects for a HIS system, e.g., *OldMedicalRecords, RecentMedicalRecords, Prescriptions, PatientMedicalInfo, PatentientPersonalInfor*, or *LegalAgreement*. Roles may access these objects by a couple of different access operations such as *view, add, modify, create*, or *sign*. For example, the role *Doctor* may *create Prescriptions*, while the role *Nurse* may only *view Prescriptions*.

The entire set of policy elements that are used in this dissertation, e.g., the role-permission assignment (Table B.1), is illustrated in Appendix B.

3.4.5 RBAC Model for the HIS Policy

The goal of re-engineering the RBAC$_3$ model is to show that RBAC models also share design principles that allow to specialize the common model core with low effort. For this purpose, we demonstrate that core-based model engineering of a dynamic RBAC$_3$ model is straightforward. Exposing the model core results in a model that fulfills the requirements of the sample policy in Section 3.4.4.

In principle, the model components (step 1) are analogous to the RBAC$_3$ definition with only minor changes:

- $U, R, S, UA, \mathit{user}, \mathit{roles}, O, OP, RH$, and RE are unchanged in comparison to RBAC$_3$.
- To emphasize the affinity of access control matrices and permissions, we define the role permission assignment relation PA as an ACM $m : R \times O \to 2^{OP}$ denoting that a role has the right to perform a specific set of operations on an object.

Step 2 selects all model components that are modifiable by primitive actions. According to our health care policy, dynamic components are: U, S, UA, user, and roles. The state set of the model core is thus defined as $Q = 2^U \times 2^S \times 2^{UA} \times USER \times ROLES$. Accordingly, a model state is defined as $q = (U_q, S_q, UA_q, user_q, roles_q)$ where $U_q \subseteq U, S_q \subseteq S, UA_q \subseteq UA, user_q \in USER = \{user \mid user : S \to U\}$, and $roles_q \in ROLES = \{roles \mid roles : S \to 2^R\}$.

In step 3 we specialize the core's extension vector. All components that are not part of the model state, are static and can now be included in the extension vector: m, R, O, OP, RH, and RE. Thus, the extension vector is 6-dimensional $(Q, \Sigma, \delta, q_0, (O, OP, R, m, RH, RE))$ and each element is defined as above.

Having defined all the state and extension vector components, we can now derive the input vector set X in step 4. According to the specialized Q, q, and E, the type set is defined as $T = \{2^U, 2^S, 2^{UA}, USER, ROLES, U, S, UA, user, roles, O, OP, R, m, RH, RE, 2^O, 2^{OP}, 2^R, 2^m, 2^{RE}\}$ with $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$.

To modify the model state, step five defines primitive actions that allow for creating any model state needed by a policy. Thus, the model contains a total of ten primitive actions that perform pairwise modifications on the state components. For example, there are two primitives modifying UA: *assignUserToRoles* and *revokeUserFromRoles*. A list of all primitive actions is given in Appendix A.3.

$$\begin{aligned}
&assignUserToRoles : Q \times X \to Q\\
&assignUserToRoles(q, x_u, \{x_{r_1}, \ldots, x_{r_n}\}) \mapsto (U_q, S_q, UA_q \cup \{(x_u, x_{r_w}), \ldots, (x_u, x_{r_n})\},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad user_q, roles_q)
\end{aligned}$$

$$\begin{aligned}
&revokeUserFromRoles : Q \times X \to Q\\
&revokeUserFromRoles(q, x_u, \{x_{r_1}, \ldots, x_{r_n}\}) \mapsto (U_q, S_q, UA_q \backslash \{(x_u, x_{r_w}), \ldots, (x_u, x_{r_n})\}
\end{aligned}$$

It remains to define the conditions for the model in step six. One condition is already given by the standard RBAC$_3$ model in terms of the *access*-function. Hence, it only needs to be rewritten in core notation:

$$cond_{core} : Q \times X \to BOOL \text{ where}$$

$$cond_{core}(q, x_s, x_o, x_{op}) \mapsto \begin{cases} true, & \exists r, r' \in R : r \succeq r' \wedge r' \in roles_q(s) \\ & \wedge\, x_{op} \in m(r', x_o) \\ false, & \text{otherwise} \end{cases}$$

$cond_{core}$ consists of three clauses and each clause is rewritten in functional notation like:

$$clause_{RH} : X \to BOOL$$

$$clause_{RH}(x_{r_1}, x_{r_2}) \mapsto \begin{cases} true, & x_{r_1} \succeq x_{r_2} \\ false, & \text{otherwise} \end{cases}$$

$$clause_{UA} : Q \times X \to BOOL$$

$$clause_{UA}(q, x_r, x_u) \mapsto \begin{cases} true, & (x_u, x_r) \in UA_q \\ false, & \text{otherwise} \end{cases}$$

$clause_{RH}$ is responsible for checking whether two roles are related in the role hierarchy; $clause_{UA}$ queries the model state whether a user is assigned a specific role. The dynamic RBAC model contains five conditions (Appendix A.3) where one condition implements separation of duty and three conditions are deviations of $cond_{core}$. Additionally, the model defines a total of five clauses.

Summary We have shown that even though static RBAC models do not contain the model core, they share design principles that enable core-based model engineering. The challenge of re-engineering a static model has been to determine the components of the model state. This is usually provided by the policy that the model is designed for. Once dynamic and static components are defined, re-engineering an RBAC model is straightforward since many components are already defined by the static RBAC model, e.g., conditions and clauses, that just need to be rewritten in core notation. The result is a dynamic RBAC model reusing the abstractions of the static RBAC_3 model.

The outcome of this section is that a wide spectrum of RBAC security policies is now unlocked to a range of engineering methods and tools for model engineering, model analysis and model implementation. This contributes to exploiting the potential of RBAC models.

3.4.6 Attribute-based Access Control Models

At the end of the 1990s, industrial and government organizations alike increasingly required collaborative business processes beyond organizational boundaries. This requirement has inspired loosely coupled systems with a high degree of interoperability that use web services as system interfaces. The increasing use of web services was attended by a need for new access control models. In contrast to traditional IT systems where the set of users is well-known and changes only sporadically, the user set of web services in service-oriented architectures is usually not known a priori and may change ad-hoc. Consequently, IBAC or RBAC models are not feasible to formalize access control policies for such highly interoperable systems. Instead, a user's properties such as her age, her location, or the organization she belongs to, now come to the fore. Besides that, [251] argue that traditional access control models are too simple, static, and coarse-grained to model security policies for web service applications. As a result, a wide variety of attribute-based access control (ABAC) models, e.g., [37,39,63,130, 147,193,219,251,254] and flexible access control frameworks supporting attributes [32,245] have been developed. However, although numerous ABAC models have been published, there is no widely accepted ABAC model like the NIST standard for RBAC models yet [130].

In general, ABAC models are characterized by access control decisions that are based on subject and object attributes rather than subject identity or roles. By means of two attribute assignment functions $as : Subject \to Att_1 \times Att_2 \times \ldots \times Att_n$, $n \in \mathbb{N}$, and $ao : Object \to Att_1 \times Att_2 \times \ldots \times Att_m$, $m \in \mathbb{N}$, attribute values are assigned to all subjects and objects. In order to make access control decisions, subject and object attribute values then are traded against each other, depending on the attributes' syntax and semantics. For example, in order to access data via web services, a user's location within the network of an organization may be critical: if a user is in a local area network (LAN) of an organization, she may access all objects with all security levels. If she is in a wide area network (WAN), however, she may only access objects that have the attribute value 'public' as security level. The user's identity here is irrelevant; only her attributes reflecting her location are important.

Unfortunately, many ABAC models, e.g., [37,39,63,147,219,251], have the same drawback as Sandhu's RBAC models (Section 3.4.3): they only specify a static snapshot of an ABAC policy without considering runtime modifications of this snapshot. However, recent work such as [59,130,245] shows that modifying subject and object attributes are key features of ABAC systems. Additionally, these features are also required to analyze ABAC models regarding their safety properties as discussed by [254]. Thus, to exploit the potential of ABAC models, first work like [59,130,254] has upgraded static ABAC models by integrating model dynamics.

We take this work one step further: our approach is to structurally engineer a dynamic

ABAC model by means of the ABAC model for web services introduced by [251]. We have chosen this model, because it is very general and can be adapted to support a wide variety of ABAC policies. The goal is to demonstrate that ABAC models also share the common model core, even though they are static by design. For this purpose, this section gives a short overview of the original ABAC model. In doing so, we demonstrate the adaptation of the general model by means of an example policy for an Online Entertainment Store inspired by [10, 251]. The Online Entertainment Store provides web services for watching movies; it has to guarantee that users may only watch those movies whose content rating approves a user's age. Section 3.4.7 then re-engineers the adapted ABAC model as core-based ABAC model.

In its most general form the original ABAC model introduced by [251] defines three basic sets: subject set S, set of resources R, and set of environments E, and for each set there is an attribute assignment function a_S, a_R, and a_E, which maps attribute values to subjects, resources, or environments respectively.[4] Subject attributes define the characteristics of a subject; for example, such attributes may be a subject's name, organization, job title, or role. Resources are entities that subjects act upon; their attributes are usually extracted from their metadata, e.g., the title, date, or author of a resource. Environment attributes describe the operational, technical, and situational environment or context, in which the information access occurs [251]. Such attributes can be the current date, time, or the network's security level.

- $a_S : S \to SA_1 \times SA_2 \times \ldots \times SA_K$, where SA_k are the pre-defined attributes for subjects, $1 \leqslant k \leqslant K$,
- $a_R : R \to RA_1 \times RA_2 \times \ldots \times RA_M$, where RA_m are the pre-defined attributes for resources, $1 \leqslant m \leqslant M$, and
- $a_E : S \to EA_1 \times EA_2 \times \ldots \times EA_N$ where EA_n are the pre-defined attributes for environments, $1 \leqslant n \leqslant N$.

Given all attribute assignments, access decisions are then made by a set of *policy rules*, which are Boolean functions evaluating the attributes of some combination of s, r, and e. A policy rule is generally defined as:

$$can\_access : S \times R \times E \to Bool$$

$$can\_access(s, r, e) \mapsto f(a_S(s), a_R(r), a_E(e))$$

A subject may access a resource in a given environment if the evaluation of some function f is true. Otherwise access is denied. We now adapt this general model to an ABAC model that is able to formalize the security policy of an Online Entertainment Store; here, we also provide examples for policy rules using specific attributes.

In the context of the Online Entertainment Store, only two of the basic sets are needed to express a suitable security policy: the set of subjects S, acting on behalf of the users, and the set of resources R, representing the movies. The environment set E along with environment attributes is not explicitly addressed in this example; though, [251] shortly discusses

[4]Note that we deviate from the original model here: it defines attribute assignment relations instead of functions. We use the functional notation, because it is more convenient for defining clauses and primitive actions in the following section. This is feasible since the relations must be left-total and right-unique. Besides, the original model also uses the functional notation for accessing value assignments.

how environment attributes may be integrated in this application scenario. Critical subject attributes are a user's age and her role, indicating whether a user is a *premium* or a *regular* customer based on the membership fee paid. Thus, the set of subject attributes consists of two elements $SA = \{AGE, ROLE\}$, where $AGE \subseteq \mathbb{N}$ and $ROLE = \{Premium, Regular\}$. Resource attributes important for this application scenario are the content rating of a movie and its release date. The latter allows to categorize movies either as *New Release* or *Old Release*. Thus, the set of resource attributes also consists of two elements $RA = \{RATING, TYPE\}$, where $RATING = \{R, PG-13, G\}$ represents the movie ratings and $TYPE = \{NewRelease, OldRelease\}$ contains the release types. The authorization rules for movie ratings and user age are shown in Table 3.1.

Movie Rating	**Authorized Users**
R	Age 21 or older
$PG-13$	Age 13 or older
G	Everyone

Table 3.1: Authorization Rules [251]

Based on these subject and resource attributes, the attribute assignment functions are defined as $a_S : S \to AGE \times ROLE$ and $a_R : R \to RATING \times TYPE$.

In order to make policy decisions, the ABAC model defines two policy rules based on the attribute assignment functions. can_access_{AR} evaluates a movie's content rating with respect to a user's age by enforcing the authorization rules of Table 3.1. can_access_{RT} evaluates a user's role with respect to a movie's release type such that *regular* customers may only watch *Old Releases* of a specific movie, while *premium* users may watch all movies. Due to convenience reasons, we use the functional notation $age(s)$ and $role(s)$ to individually select the first or second tuple element of the subject attribute assignment $a_S(s) \mapsto (age(s), role(s))$. The individual attribute values of a resource are selected by $rat(r)$ and $type(r)$ where $a_R(r) \mapsto (rat(r), type(r))$.

$$can\_access_{AR} : S \times R \to BOOL$$

$$can\_access_{AR}(s,r) \mapsto \begin{cases} true, & (age(s) \geqslant 21 \wedge rat(r) \in \{R, PG, G\}) \\ & \vee\ (age(s) \geqslant 13 \wedge age(s) < 21 \wedge rat(r) \in \{PG, G\}) \\ & \vee\ (age(s) < 13 \wedge rat(r) \in \{G\}) \\ false, & \text{otherwise} \end{cases}$$

$$can\_access_{RT} : S \times R \to BOOL$$

$$can\_access_{RT}(s,r) \mapsto \begin{cases} true, & (role(s) = \text{`}Premium\text{'}) \\ & \vee\ (role(s) = \text{`}Regular\text{'} \wedge type(r) = \text{`}OldRelease\text{'}) \\ false, & \text{otherwise} \end{cases}$$

The access function $can\_access : S \times R \to \text{BOOL}$ is then defined by conjunctively combining both policy rules such that

$$can\_access(s,r) \mapsto can\_access_{AR} \wedge can\_access_{RT}$$

By means of these model elements, a static snapshot of an ABAC policy for an Online Entertainment Store can be formalized. The following section now demonstrates that even though this ABAC model is static, it is based on design principles that allow for re-engineering it as core-based model that supports policy dynamics. Note that the re-engineering method can also be applied for the ABAC model in its most general form and any specific ABAC model that is derived.

3.4.7 ABAC Model for an Online Entertainment Store Policy

The goal of this section is to demonstrate that core-based engineering of an ABAC model is feasible and straightforward. On top of that, we extend the static ABAC model [251] by introducing policy dynamics that is expressed in core notation. For this purpose, this section re-engineers the adapted ABAC model that formalizes a static policy of an Online Entertainment Store as core-based ABAC model (Section 3.4.6). In doing so, we show that the adapted ABAC model and also the general ABAC model of [251] are based on design principles that render core-based model engineering possible.

The first step is to determine the components of the core-based ABAC model: $M = \{S, R, a_S, a_R, SA, RA, AGE, ROLE, RATING, TYPE\}$. As can be seen, these are identical to the components of the adapted security model with $SA = \{AGE, ROLE\}$ and $RA = \{RATING, TYPE\}$.

The next step defines the model state, which is provided by the application scenario of the Online Entertainment Store. Here, key features are to add and delete users and movies, and to change their attribute assignments, for instance if a user gets older or the release type of a movie changes. Thus, dynamic model components are S, R, a_S, and a_R such that the state set is defined as $Q = 2^S \times 2^R \times A_S \times A_R$ where $A_S = \{a_s \mid a_S : S \to AGE \times ROLE\}$ and $A_R = \{a_R \mid a_R : R \to RATING \times TYPE\}$. Hence, any state is a tuple $q = (S_q, R_q, a_{S_q}, a_{R_q})$ with $S_q \subseteq S$, $R_q \subseteq R$, $a_{S_q} \in A_S$, and $a_{R_q} \in A_R$.

From this it follows that the extension vector E is 2-dimensional, containing the components SA and RA. The core-based ABAC model hence is defined as $(Q, \Sigma, \delta, q_0, (SA, RA))$.

Based on the definition of the model state and the extension vector, we can now specify the input types T of the input vector set X in the fourth step. With respect to Q, q, and E, T is defined as $T = \{2^S, 2^R, A_S, A_R, S, R, a_S, a_R, AGE, ROLE, RATING, TYPE, 2^{AGE}, 2^{ROLE}, 2^{RATING}, 2^{TYPE}\}$.

The fifth step defines the set of primitive actions for modifying the model state. As already mentioned, the Online Entertainment Store features the modification of the subject and the resource set, requiring that the subject and object attribute assignments can also be modified. On this account, a total of eight primitive actions is needed that pairwise modify the state components. For example, the subject set may be modified by $addSubject : Q \times X \to Q$ and $deleteSubject : Q \times X \to Q$; the subject attributes can be modified by $assignSubjAttributes : Q \times X \to Q$ and $resignSubjAttributes : Q \times X \to Q$. An overview of all primitive actions of the ABAC model can be found in Appendix A.4.

$$addSubject : Q \times X \to Q$$
$$addSubject(q, \{x_s\}) \mapsto (S_q \cup \{x_s\}, R_q, a_{S_q}, a_{R_q})$$

$$deleteSubject : Q \times X \to Q$$
$$deleteSubject(q, \{x_s\}) \mapsto (S_q \setminus \{x_s\}, R_q, \{s\} \lhd a_{S_q}, a_{R_q})$$

$$assignSubjAttributes : Q \times X \to Q$$
$$assignSubjAttributes(q, x_s, x_{age}, x_{role}) \mapsto (S_q, R_q, a_{S_q} \oplus \{(x_s, x_{age}, x_{role})\}, a_{R_q})$$

$$resignSubjAttributes : Q \times X \to Q$$

$$\mathit{resignSubjAttributes}(q, x_s, x_{age}, x_{role}) \mapsto (S_q, R_q, a_{S_q} \backslash \{(x_s, x_{age}, x_{role})\}, a_{R_q})$$

The last step specifies the conditions of the ABAC model. One condition is already given by the static ABAC model by the policy rules $\mathit{can\_access}_{AR}$ and $\mathit{can\_access}_{RT}$. It only remains to rewrite them as clauses and combine them by the condition cond_{access}. The rewriting of $\mathit{can\_access}_{AR}$ is shown in the following; rewriting $\mathit{can\_access}_{RT}$ is illustrated in Appendix A.4.

$$\mathit{clause}_{AR} : Q \times X \rightarrow \mathit{BOOL}$$

$$\mathit{clause}_{AR}(q, x_s, x_r) \mapsto \begin{cases} \mathit{true}, & (\mathit{age}_q(x_s) \geqslant 21) \\ & \vee\ (\mathit{age}_q(x_s) \geqslant 13 \wedge \mathit{age}_q(x_s) < 21 \wedge \mathit{rat}_q(x_r) \in \{PG, G\}) \\ & \vee\ (\mathit{age}_q(x_s) < 13 \wedge \mathit{rat}_q(x_r) \in \{G\}) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\mathit{cond}_{access} : Q \times X \rightarrow \mathit{BOOL}$$

$$\mathit{cond}_{access}(q, x_s, x_r) \mapsto \begin{cases} \mathit{true}, & \mathit{clause}_{AR}(q, x_s, x_r) \wedge \mathit{clause}_{RT}(q, x_s, x_r) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

Since the original ABAC model is static, it is not concerned with policy administration. However, the core-based ABAC model for the Online Entertainment Store example requires a condition that ensures that the model state can only be modified by a privileged instance such as an administrator. For this reason, we have enhanced the value range of the subject attribute *ROLE* by an additional role that we call 'Admin', which can now be evaluated for state modifications by the additional management condition cond_{man}. In case this is too course-grained, additional attribute values may be added according to a given policy specification.

$$\mathit{clause}_{man} : Q \times X \rightarrow \mathit{BOOL}$$

$$\mathit{clause}_{man}(q, x_s) \mapsto \begin{cases} \mathit{true}, & \mathit{role}_q(x_s) = \text{`}\mathit{Admin}\text{'} \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\mathit{cond}_{man} : Q \times X \rightarrow \mathit{BOOL}$$

$$\mathit{cond}_{man}(q, x_s, x_r) \mapsto \mathit{clause}_{man}(q, x_s)$$

The core-based ABAC model altogether defines a total of three clauses and two conditions, out of which two clauses and one condition have already been predefined by the original ABAC model. The condition cond_{man} based on clause_{man} has been added for guarding state modifications via primitive actions.

Summary This section has shown that although ABAC models are not originally based on the model core, they share design principles that render core-based model engineering possible. As with static RBAC models (Section 3.4.5), the challenge of re-engineering an ABAC model is to determine the components of its state. Once dynamic and static components are defined, the re-engineering is straightforward since many components are already specified by the static ABAC model and just need to be rewritten in core notation. For example, the original ABAC model defines two policy rules, which can easily be rewritten as two clauses. Result is a dynamic ABAC model that applies the abstractions of the static ABAC model enriched by the core.

This section's achievement is that numerous ABAC models, namely all those that can be derived from the general ABAC model of [251], are now unlocked to a variety of methods and tools for model engineering, analysis, and implementation. This makes a significant contribution to developing and exploiting the potential of a standard ABAC model.

3.4.8 Summary

We have illustrated core-based model engineering by means of three models: an MLS, an RBAC$_3$, and an ABAC model. We have argued that these models are relevant for specifying, analyzing, and implementing real-world security policies, since there have been numerous application scenarios and IT systems implementing them.

Even though these models belong to different domains and contain domain-specific model abstractions (e.g., lattices, role hierarchies, and attribute assignments), they also share common abstractions, namely the common core that we have exposed by applying core-based model engineering. This section has shown that the approach of a domain-independent model core is feasible and universal. It can be applied for models belonging to the access control or the information flow domain. Moreover, core-based model engineering is comprehensible and straightforward since it is composed of six clearly arranged steps that build upon each other.

The main merits of core-based model engineering are twofold: (i) models in core notation are able to express policy dynamics by means of the deterministic automaton. (ii) Core-based models share common abstractions, which is the key enabler for applying model-independent methods for model engineering, analysis, implementation, and execution. On this account, core-based model engineering contributes to an integrated security policy engineering process without semantic gaps.

3.5 Model Core Evaluation

The goal of this section is to evaluate the model core along with core-based model engineering with respect to its expressive power and the resulting modeling costs. For this purpose, we first discuss the types of models that can be engineered with core-based model engineering in Section 3.5.1. Afterwards, we consider the modeling costs that come with core-based model engineering (Section 3.5.2).

3.5.1 Expressive Power

The common model core is a deterministic automaton that generalizes the ideas of the HRU model. Thus, its computational power is precisely equivalent to a Turing Machine. The computational power alone, however, is not sufficient to evaluate the expressive power of a security model, which captures the notion of whether different policies can be adequately formalized by that model [234]. A security model is said to express a policy adequately if it provides model abstractions with only a minimal semantic gap to the policy's paradigms. On this account, we subsequently discuss the expressive power of the model core with respect to different policies and their models.

In this context, the four *Role-based Trust management (RT)* models $\mathsf{RT}[\,]$, $\mathsf{RT}[\cap]$, $\mathsf{RT}[\twoheadleftarrow]$, and $\mathsf{RT}[\cap, \twoheadleftarrow]$ introduced by [156,159] take a special position. In contrast to the HRU model, the authors have shown that safety in their models is efficiently decidable [156,159]. For this reason, they have studied the relationships between the RT models and the HRU model with the result that the HRU model cannot adequately express the RT models [156,159]. On this account, the goal of this section is to show that core-based model engineering is able to even produce a model whose expressive power is incomparable to the expressive power of the HRU model [156, 159, 234].

Section 3.3 has illustrated that the model core still subsumes the HRU model, which has been generally believed to have considerable expressive power [107, 108, 233, 234]. Consequently, the model core must be at least as expressive as the HRU model. So far, we have supported this claim by showing that the model core can engineer a wide variety of DAC and MAC models, namely IBAC, RBAC, and ABAC models (Sections 3.3 and 3.4). Additionally, we have demonstrated how to engineer an MLS model as a prominent representative for information flow models.

Recent work has shown that the expressive power of the HRU model is limited though. More precisely, while [156, 159] have informally discussed that already a rather simple trust-management model, RT[ ], cannot be adequately expressed by an HRU model, [234] has given the first formal evidence. On the other hand, [156, 159] have discussed that RT[ ] also cannot subsume the HRU model due to its complexity bounds. The reason are the different commands supported by the HRU model and RT[ ]. Thus, the expressive powers of the HRU and the RT[ ] model are mutually exclusive.

From the formal proof of [234] follows: if we can derive a model from the model core that is expressively equivalent to RT[ ], we show that the model core is more expressive than the HRU model. However, thereby we also show that the model core is more expressive than RT[ ], since (i) the model core can be specialized as an HRU model (demonstrated in Section 3.3) and (ii) RT[ ] cannot adequately express the HRU model as shown by [156, 159].

In order to verify this dissertation, the remainder of this section analyses whether RT[ ] can be expressed as a core-based model. For this purpose, we briefly discuss the characteristics of RT[ ], before sketching core-based RT[ ] model engineering.

3.5.1.1 RT[ ] Model

The goal of trust management (TM) models is to express delegation in distributed systems. A principal may transfer limited authority over one or more resources to other principals, which results in giving away a certain degree of control [156, 159]. Here, the question arises whether a resource owner still has some guarantees about who can access their resources [156, 159]. The RT models are members of the *RT* model family [153–155, 160], combining the advantages of RBAC and TM models [155]. The goal of the RT models is to model trust delegation by means of role delegation, in order to enable security analysis in trust management systems. For this purpose, the models contain a set of statements that define how roles are assigned and delegated to principals; the roles' specific permissions are irrelevant here. The authors have introduced four models RT[ ], RT[$\cap$], RT[$\twoheadleftarrow$], and RT[$\cap, \twoheadleftarrow$] [159], which together form a slightly simplified but expressively equivalent version of RT_0 [160] as the simplest member of the *RT* family. These models differ in the types of role delegation statements they support; RT[ ] is the most basic one with only two types of statements.

The basic sets of the RT models are principals, roles, and role names [159]. Principals (P) represent the entities of a TM system and are denoted by A, B, D, F, X, Y, and Z [160]. Role names (N) are identifiers denoted by r, u, and w. Roles (R) consist of a principal and a role name, separated by a dot, e.g., $A.r$, thereby assigning a role to a principal, which gains the permission to assign the role to other principals. For example, $A.r$ is a role whose member A may assign $A.r$ to other principals [159].

All RT models are based on an automaton, where each state is described by a set of delegation statements. States of RT[ ] consist of two types of delegation statements, each of which being a statement made by a principal about the members of its role:

- *Simple Member:* $A.r \longleftarrow B$
- *Simple Inclusion:* $A.r \longleftarrow B.r_1$

Simple member is a statement made by A, that defines B as a member of A'*s* role $A.r$. On the other hand, simple inclusion is a statement made by A, which says that all members of B'*s* role $B.r_1$ are also members of A'*s* role $A.r$. This represents a delegation by A to B, since only B has the authority to specify the members of its roles. In other words, $B.r_1$ contains (all the members of) $A.r$, which is the inverse of a dominance relation [155]. The left part of a statement (here $A.r$) is called the statement's head; the right part (here B or $B.r_1$) is called the statement's body. Refer to [155] for more details on RT statements.

State transitions in RT[ ] are made by adding and removing simple member and simple inclusion statements, resulting in the modification of a state's set of statements. State transitions are guarded by state-independent *restriction rules* of the form $\mathcal{R} = (\mathcal{G}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}})$ where $\mathcal{G}_{\mathcal{R}} \subseteq R$ and $\mathcal{S}_{\mathcal{R}} \subseteq R$ are sets of RT[ ] roles. $\mathcal{G}_{\mathcal{R}}$ is the set of *growth-restricted* roles; no policy statement assigning these roles can be added [159]. That means, if $A.r \in \mathcal{G}_{\mathcal{R}}$, statements with $A.r$ as head may not be added [234]. Roles in $\mathcal{S}_{\mathcal{R}}$ are called *shrink-restricted*; policy statements that assign these roles cannot be removed [159]. That means, if $A.r \in \mathcal{S}_{\mathcal{R}}$, statements with $A.r$ as head may not be removed [234].

[159] defines three kinds of queries in RT[ ] for analyzing a model's security properties. Analysis problems are answered by evaluating these queries with logic programs that are derived from the model state $\mathcal{P}$ and the restriction rule $\mathcal{R}$. For more details refer to [156,159].

3.5.1.2 RT[ ] Model Re-engineering

Domain-specific security models such as RBAC and ABAC models, and particularly the RT models [156, 159] have identified the following differences compared to HRU models:

1. In RT[ ], creating and removing principals are implicit. A principal can be understood as created if it is used in a statement; a principal is considered removed if there is no statement that contains it. In contrast, the HRU model requires that all subjects and objects are explicitly created and removed by using the primitive operations *create/destroy subject* and *create/destroy object*.

2. One atomic state change operation in RT[ ] corresponds to executing many primitive operations in the HRU model, depending on the number of subjects or objects in a matrix. The primitive section of HRU commands, however, does not support universal or existential quantifiers.

3. The HRU model does not provide any means to model delegation as defined by RT[ ]. To support delegation, auxiliary constructs such as trigger programs would be required. For example, adding $A.r \longleftarrow B.r_1$ means that A delegates B the authority to delegate A'*s* role r to all members of $B.r_1$. If another statement such as $B.r_1 \longleftarrow E$ is added afterwards, E becomes a member not only of B'*s* role r_1 but also of A'*s* role r. To enforce the latter role assignment, trigger programs are needed that must be executed whenever the matrix is modified.

In the following, we briefly sketch core-based model engineering of RT[ ], and in doing so we show that the model core does not share these shortcomings.

The set of components of RT[ ] is defined as $M = \{P, N, R, mem, RH, \mathcal{G}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}}\}$ where P, N, R, $\mathcal{G}_{\mathcal{R}}$, and $\mathcal{S}_{\mathcal{R}}$ are defined as above, $mem : R \rightarrow 2^P$ is a role-to-member assignment function, and $RH \subset R \times R$ is a role hierarchy, modeled by a partial order on R, written as $\succeq$, where $\forall r, r' \in R : r \succeq r' \Leftrightarrow r$ dominates r', or any user who is a member of r also is automatically a member of r' [155]. Hence, $mem(A.r) \mapsto \{B\}$ represents a role assignment by a simple member statement $A.r \longleftarrow B$ in core notation, and a simple inclusion statement $A.r \longleftarrow B.r_1$ is represented by $(B.r_1, A.r) \in RH$ where $B.r_1 \succeq A.r$.

A state transition in RT[ ] is modeled by adding or removing at least one simple member or one simple inclusion statement. In the equivalent core-based RT[ ] model, this corresponds to modifying mem and RH, including R and N since they contribute to the domain of mem. P is modified whenever principals are added or removed. Thus, the state set is defined as $Q = 2^P \times 2^N \times 2^R \times MEM \times 2^{RH}$ where $MEM = \{mem | mem : R \rightarrow 2^P\}$; any RT[ ] state in core notation is defined as $q = (P_q, N_q, R_q, mem_q, RH_q)$ where $P_q \subseteq P$, $N_q \subseteq N$, $R_q \subseteq R$, $mem_q \in MEM$, and $RH_q \subseteq RH$. The extension vector E consists of the remaining elements of M such that $E = (\mathcal{G}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}})$ and the model is defined by $(Q, \Sigma, \delta, q_0, (\mathcal{G}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}}))$.

Since adding or removing statements are the most primitive changes of the state, these changes must be implemented by the set of primitive actions. Hence, the model defines four primitive actions, which we call *add_simple_member* (*asm*), *remove_simple_member* (*rsm*), *add_simple_inclusion* (*asi*), and *remove_simple_inclusion* (*rsi*). Adding a statement $A.r \longleftarrow B$ equals calling the primitive action $asm(q, A.r, B)$, respectively $rsm(q, A.r, B)$ for removing it. Adding or removing a statement $A.r \longleftarrow B.w$ corresponds to calling $asi(q, A.r, B.w)$ or $rsi(q, A.r, B.w)$:[5]

$$\begin{aligned}
&asm : Q \times R \times P \rightarrow Q\\
&asm(q, A.r, B) \mapsto (P_q \cup \{A, B\}; R_q \cup \{A.r\}; N_q \cup \{r\}; \forall X.u \in R_q,\\
&\qquad X.u \in devotee(q, A.r) : mem_q \oplus \{(X.u, mem_q(X.u) \cup \{B\})\}; RH_q)
\end{aligned}$$

$$\begin{aligned}
&rsm : Q \times R \times P \rightarrow Q\\
&rsm(q, A.r, B) \mapsto (\forall p \in P_{IN}, \nexists(X.u, Y.w) \in RH_q, X = p \vee Y = p, \nexists(X.u, P_1) \in mem_q,\\
&\qquad p \in P_1 : P_q \backslash \{p\}; R_q; N_q; \forall X.u \in R_q, X.u \in devotee(q, A.r) :\\
&\qquad mem_q \backslash \{(X.u, mem_q(X.u))\} \cup \{(X.u, mem_q(X.u) \backslash \{B\})\}; RH_q)
\end{aligned}$$

$$\begin{aligned}
&asi : Q \times R \times R \rightarrow Q\\
&asi(q, A.r, B.w) \mapsto (P_q \cup \{A, B\}; R_q \cup \{A.r, B.w\}; N_q \cup \{r, w\}; \forall X.u \in R_q,\\
&\qquad X.u \in devotee(q, A.r) : mem_q \oplus \{(X.u, mem_q(X.u)\\
&\qquad \cup\, mem_q(B.w))\}; RH_q \cup \{(B.w, A.r)\})
\end{aligned}$$

$$\begin{aligned}
&rsi : Q \times R \times R \rightarrow Q\\
&rsi(q, A.r, B.w) \mapsto (\forall p \in P_{IN}, \nexists(X.u, Y.w) \in RH_q, X = p \vee Y = p,\\
&\qquad \nexists(X.u, P_1) \in mem_q, p \in P_1 : P_q \backslash \{p\}; R_q; N_q; \forall X.u \in R_q,\\
&\qquad X.u \in devotee(q, A.r) : mem_q \backslash \{(x.U, mem_q(X.u))\}\\
&\qquad \cup \{(X.u, mem_q(X.u) \backslash mem_q(B.w))\}; RH_q \backslash \{(B.w, A.r)\})
\end{aligned}$$

[5]For better readability state components are separated by semicolons instead of commas in the following, e.g., $q = (P_q; N_q; R_q; mem_q; RH_q)$.

We subsequently discuss a few details regarding the primitives' semantics: first, $devotee : Q \times R \rightarrow 2^R$ is an auxiliary function that recursively assembles all roles that the given role dominates (Algorithm 1). It ensures reflexivity and transitivity of the dominance relation and is thus required every time a role assignment or a delegation is made. Second, principals are added with every added statement. There are no redundancies, because this is modeled by the set operator '$\cup$'. In contrast, a principal is removed only if there is no pair both in mem_q and RH_q, containing this principal. This is enforced whenever a statement is removed. To our knowledge, roles and role names do not need to by removed. Last, P_{IN} is an auxiliary set that collects all principals of the input parameters; here it is defined as $P_{IN} = \{A, B\}$.

Algorithm 1: Dominance Relation Assembly

```
Input: a model state $q \in Q$, a role $A.r \in R$
Output: a set of roles $R_{DEV}$

function devotees(in $q \in Q$, in $A.r \in R$)
    for $X.u \in R_q$ do
        if $(A.r, X.u) \in RH_q$ then
            $R_{DEV} \leftarrow R_{DEV} \cup \{X.u\}$;
            devotees($q, X.u$);

$R_{DEV} \leftarrow \emptyset \cup \{A.r\}$;
devotees($q, A.r$);
```

It remains to specify the model's clauses, which are directly defined by the restriction rules of RT[ ] as follows: if $A.r \in \mathcal{G}_{\mathcal{R}}$, the primitive actions $asm(q, A.r, B)$ and $asi(q, A.r, B.w)$ may not be executed with $A.r$ as second input parameter. Thus, there must be a clause, called $clause_G(A.r)$, which enforces growth restriction by guarding the execution of $asm(q, A.r, B)$ and $asi(q, A.r, B.w)$. In analogy, there must be a second clause, called $clause_S(A.r)$, which enforces shrink restriction by guarding the execution of $rsm(q, A.r, B)$ and $rsi(q, A.r, B.w)$:

$$clause_G : R \rightarrow BOOL$$

$$clause(A.r) \mapsto clause_{AR} \mapsto \begin{cases} true, & A.r \notin \mathcal{G}_{\mathcal{R}} \\ false, & \text{otherwise} \end{cases}$$

$$clause_S : R \rightarrow BOOL$$

$$clause(A.r) \mapsto clause_{AR} \mapsto \begin{cases} true, & A.r \notin \mathcal{S}_{\mathcal{R}} \\ false, & \text{otherwise} \end{cases}$$

Summary This section has shown that the model core can adequately express RT[ ]. Hence, the model core does not share the expressive limits of the HRU model. The reason is that primitive actions may define arbitrary programs instead of fixed schemas as specified by the HRU model. This allows for modeling primitives in such a way that they exactly represent the atomic commands of RT[ ]: they enable the implicit adding and removing of principals and that one step of state change in RT[ ] exactly represents one step of state change in the core-base model. Besides, auxiliary constructs like trigger programs are not needed since primitive actions may contain recursive functions to model RT[ ] delegation. Beyond that, the model core can also express the other models RT[$\cap$], RT[$\twoheadleftarrow$], and RT[$\cap, \twoheadleftarrow$] of the RT family, since these merely add other types of statements.

In the previous sections, we have demonstrated that the model core also is more expressive than RT[ ]: it can still express the HRU model and a wide variety of traditional IBAC, RBAC, and ABAC models (each in their DAC and MAC variants), and MLS models as a prominent representative of information flow models.

3.5.2 Model Engineering Costs

After having discussed the expressive power of the model core, the goal of this section is to discuss the costs of core-based model engineering. Note that the notion of modeling costs is not a standard one; it is inspired by the work of Bertino et al. [32, 34], which captures structural differences of access control models to reason about the models' expressive power. Based on these ideas, we use structural properties of security models in a more general way to informally appraise the modeling effort of core-based model engineering.

Before discussing the costs of core-based model engineering in detail, there are two general statements to be made: (i) as shown in Section 3.2, the core $(Q, \Sigma, \delta, q_0, E)$ provides a modeling framework for a wide variety of dynamic access control and information flow models, which is minimal with respect to domain-specific model abstractions. Any domain-specific model abstraction, e.g., role hierarchies, matrices, or attribute assignments, along with specific model components must thus be individually defined in the course of core-based model engineering. However, this can be supported by model engineering tools such that the modeling effort is limited to specializing the model core. (ii) The actual effort involved in core-based model engineering depends on the individual goal of a modeler. Here, we differ between three engineering goals, for each of which we discuss the modeling effort in the following:

1. Re-engineering a dynamic model.
2. Re-engineering a static model.
3. Engineering a novel model.

Re-engineering a Dynamic Model If the goal is to re-engineer an existing dynamic model in core notation, every required detail is already predefined. This means, domain-specific model abstractions along with specific model components do not have to be engineered, and the composition of a deterministic automaton and the components is also already specified (by means of the model state and state transition rules), even if it is done in an informal way. Hence, it only remains to find an adequate representation in core notation for all components; the semantic composition of the components and the automaton does not change. Out of the three goals, this is the easiest, resulting in the lowest modeling effort, which only arises from component rewriting.

For example, Section 3.5.1 has demonstrated this kind of model re-engineering based on RT[ ]. Here, all model components $(P, R, N, A.r \longleftarrow B, \dots)$, the model state as well as state transition and restriction rules $(\mathcal{R} = (\mathcal{G}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}}))$ have been defined, though partially informally. The remaining challenge has been to rewrite the model components in core notation, i.e., to find a precise representation of the simple member and simple inclusion statements along with primitive actions. Besides, the model clauses have been directly adopted from the restriction rules $\mathcal{R} = (\mathcal{G}_{\mathcal{R}}, \mathcal{S}_{\mathcal{R}})$.

Re-engineering a Static Model The modeling effort for re-engineering a static model is very low, since only the extension vector E of the model core has to been specialized.

On top of that, the model core provides the opportunity to enhance a static model by dynamic model components, which results a security model that can express policy dynamics. However, this requires that the remaining core components are also specialized. Compared to re-engineering a dynamic model, this is more comprehensive and involves more modeling effort. The reason is that even though the static model already defines its components, the composition of the components and a deterministic automaton is missing. Thus, the challenge here is to define such a composition by mapping the existing model components to the components of the model core.

The required mapping steps are as follows: the model state and the extension vector have to be defined first. A modeler therefor has to know the security policy to be formalized or at least the policy's application scenario. Once she has determined the dynamic components, specializing the state and the extension vector merely deals with rewriting the corresponding components in core notation (previous paragraph). The second step is to specify the primitive actions and conditions. This step does not only semantically depend on the state and extension vector definition, but also in terms of the modeling effort: in general it holds, the more tuple elements a state contains, the more primitive actions may be specified. While the set of primitive actions needs to be created from scratch, the condition set may already be partly defined by the original model. The reason is that static models usually define authorization or information flow rules for allowing a subject to access a resource or information to flow. These rules only need to be rewritten as conditions in core notation and possibly extended by additional conditions that guard the execution of primitive actions. Examples of this kind of model engineering are given in Sections 3.4.5 and 3.4.7.

The higher modeling effort hence results from specifying the semantics of the model core components based on the predefined model components, namely by mapping the model components to q and E, and defining primitive actions and additional conditions. The modeling effort for rewriting the existing model components and conditions remains the same compared to re-engineering a dynamic model.

Engineering a Novel Model Compared to the other engineering types, a modeler has to start from scratch when engineering a novel core-based security model. That means, since the model core does not provide any domain-specific model abstractions to be used, they have to be modeled, too. Afterwards, the modeler has to derive all components that are required by the novel model: (i) entities to be protected and active entities, (ii) entity attributes, and (iii) relations that describe interrelations between entities and attributes. Here, the modeling effort depends on the complexity of a security model and its components. Once all components are defined, they can be mapped to the components of the model core's automaton (previous paragraph).

From this it follows that compared to the other two types of core-based model engineering, engineering a novel model is the most comprehensive one, involving the most modeling effort. The reason is that in addition to the modeling steps of the other types, it also requires the definition of adequate model abstractions and components, which may be very burdensome and requires at lot of modeling experience.

Furthermore, engineering a novel model in core notation usually involves more effort than when applying a domain-specific model that already provides adequate abstractions. The advantage of a model in core notation though is that core-based security model are unlocked to a range of model-independent methods and tools for model engineering, analysis, implementation, and execution.

3.5.3 Summary

This section has argued that the model core does not share the expressive limits of HRU models since it is able to adequately express the RT models as representatives of TM models. Additionally, the model core is still able to express traditional IBAC, RBAC, ABAC models (each in their DAC and MAC variant), including the HRU model, and information flow models, which shows that the model core is even more expressive than the RT models.

The basis of the model core's expressive power is a minimal modeling framework that is the key enabler of model-independent methods and tools for model engineering, analysis, implementation, and execution. Since the minimal framework does not provide domain-specific abstractions, modeling a novel core-based model leads to a higher modeling effort than when domain-specific models that already provide adequate model abstractions are applied. In case an existing model is to be re-engineered, the modeling effort depends on the model to be re-engineered: if it is dynamic and already applies a deterministic automaton, the effort is limited to rewriting its components in core notation. In case it is static, the modeling effort arises from composing the existing components with the model core in addition to rewriting the model components in core notation.

3.6 Model Core Related Work

Since 1970, the research community has proposed a wide variety of security models for expressing domain-specific access control and information flow rules. As a consequence, researchers have spent much effort in developing generalized models and languages that can express a range of models, in order to provide a basis for common policy specifications. The goal of this section is to differentiate the model core from related work.

Some of the relevant related work is roughly based on the same motivation as the model core: designing a general access control model, from which both existing and novel models can be derived. In this context, Barker's *meta-model of access control* [20–22] expressed in a fibred security language is closely related; several similar approaches are *SecPAL* [24] and the *RT* model family [155], providing policy languages that allow for expressing decentralized authorization policies. The proposed meta-model is based on a similar approach that we have adopted to identify the model core: identifying primitive notions of existing access control models that can then be specialized [22]. While our result has been the deterministic automaton combined by a vector of extensions, Barker et al. have identified primitive notions that generalize domain-specific model abstractions: principals, categories, relationships between categories and between categories and principals, and modalities (e.g., permissions and authorizations). This differs from our approach in that the model core does not provide any means of model abstractions, neither generalized nor specialized. To our knowledge, the reason is that [20–22] focus on access control models, which is the second significant difference. Regarding their expressive power, Barker et al. have shown that their meta-model can express rule-based access control models such as [23], RBAC and status-based access control (SBAC) models, DAC as well as MAC models by means of the "no read up“ and ”no write down“ rules of the BLP model [26]. On top of that, the authors have demonstrated that statements of the RT family proposed by Li et al. [155] can also be expressed by their meta-model. From this it follows, that the meta-model is able to express a wide variety of security models just as the model core.

Based on Barker's meta-model, [132] has developed an information flow control meta-model. By this means, [132] has demonstrated the flexibility and generality of Barker's meta-model. Moreover, the authors have specialized the information flow control meta-model to model the policy of a file sharing system and of the HiStar system [253]. Hence, they have also shown that the information flow control meta-model can formalize complex and realistic information flow control mechanisms. Compared to the model core, the information flow control meta-model shares the same difference as Barker's meta-model: it provides primitive notions that generalize domain-specific model abstractions.

Other related work has aimed at a unifying formalism to encompass a range of access control models and policies, in order to combine them for policy analysis or policy implementation [115]. These aims have lead to the *Policy Machine (PM)* [82,84,85,115], which is an access control framework and a system architecture for policy specification and enforcement. The approach of the authors also has been to identify a minimal set of primitives that can specify and enforce a large variety of access control models; however, the authors have focused on ABAC policies. Like Barker's meta-model, the PM is based on a set of generalized domain-specific abstractions, namely data, assignment and prohibition relations, obligations, and functions where data encompasses the basic elements like authorized users, processes, system operations, and objects. Additionally, the PM includes four abstractions called user attributes, object attributes, operation sets, and policy classes. By this means, the model core provides a smaller set of primitive components than compared to the PM. For modeling state transitions, PM uses *administrative operations*, which are parameterized procedures describing the modification of data sets or relations; *administrative commands* are parameterized sequences of administrative operations prefixed by a condition. Thus, administrative operations are semantically equivalent to the model core's primitive actions and administrative commands semantically equal the core's commands. That means, even though the basic abstractions of the PM are quite different, the modeling of state transitions applies the same notion as done by the model core. With respect to the expressive power, the authors have shown that the PM can express DAC and RBAC models, the Bell-LaPadula model [26] as well as the Chinese Wall model [41]; whether or not the PM can express the models of the RT family [155] remains undiscussed by the authors. However, due to [82], we doubt that modeling RT models with the PM is as straight forward as with the model core.

At SACMAT 2008, there was a panel discussing that RBAC may be fundamental to access control and that it may be extended to express a variety of other models and policies [81]. Moreover, it has been suggested that the basic relations of the PM are similar to that of RBAC and that some of the PM components could be applied in extending the RBAC model [81]. It has even been shown that RBAC is policy neutral and can express DAC and MAC models as well as lattice-based access control models [182,188,206,211]. In doing so, [206] has qualified that RBAC is more suitable for expressing MAC rather than DAC models. The reason is that a high number of specific roles is needed due to several roles are required for each object. This makes simulating DAC quite complex. From this it follows that even though RBAC in general is able to express DAC and MAC models, the resulting models merely simulate the underlying model and do not lead to tailored domain-specific security models. As such the simulating models are not suitable for model-independent analysis and implementation methods as domain-specific core-based security models are. Besides, it has not been discussed whether RBAC models can express formal models like the HRU, SPM [204], TAM [209], or the RT models, so that the actual expressive power of this approach remains unclear.

Due to its relevance, however, the RBAC model has been extensively investigated, e.g., by [30, 59, 83, 129, 139, 157]. An important work on RBAC is temporal RBAC (TRBAC) [30] that includes periodic enabling of roles and temporal dependencies among roles as required by some real-world applications. Based on this work, the *generalized temporal role-based access control (GTRBAC)* model [131] has been proposed, which subsumes TRBAC by allowing for the specification of a comprehensive set of temporal constraints. Compared to the model core, even though GTRBAC is more general as TRBAC, it only focuses on RBAC models and does not consider the rewriting of other security models. It thus cannot be compared to a unifying formalism for security models like the model core, the meta-model, or the PM. Besides, in the context of the model core, RBAC is only one out of numerous security models that can be derived from the model core.

More related work has been motivated by designing a language that can express a variety of access control models to allow for comparing the expressive power of these models [32, 34] or to provide a unified implementation environment for policies [126]. Bertino et al. [32, 34] have introduced a logical framework based on the C-Datalog language for expressing a variety of access control models in order to reason about their expressive power. More precisely, the framework captures differences by *access equivalence* and *structural equivalence* to compare the authorization rules and components of access control models. This approach, however, only deals with particular states of the models; it does not consider state-change rules. That means, even though the authors have shown that their framework can express DAC and MAC models as well as the RBAC and the Bell-LaPadula model, it does not formalize state change rules, which are an immanent part of core-based model engineering. Thus, the expressive power is limited to static access control models. Jajodia et al. [126] have proposed a framework, called *Flexible Authorization Framework (FAF)*, that can capture several discretionary access control models. However, already [32, 34] have shown that FAF can be represented by their framework because programs written in this language are a subset of Datalog programs.

Another general language for domain-specific security policies is the *eXtensible Access Control Markup Language (XACML)* [18] by OASIS. XACML is an XML-based language describing access control policies as well as policy decision requests and responses. Though XACML can be used to describe a range of access control policies such as RBAC or ABAC, it does not consider state change rules, which severely limits the policies that can be described and enforced. Moreover, since XACML is based on XML, it is not suitable for policy analyses that require access control models with a sound formal foundation.

3.7 Conclusion

This section has identified basic, domain-independent model abstractions that are shared by a large number of access control and information flow models. This sets the course for building a security model family tree beyond domain boundaries that exposes common properties of security models. The result is the identification of model abstractions shared by numerous models, which we have generalized by the uniform model core.

The model core provides the basis for re-engineering existing and engineering novel models with shared abstractions. This can be exploited in four ways: (i) reuse of model components, (ii) model-independent security analysis and implementation methods, (iii) model-independent tool support for model engineering, analysis, and implementation, and (iv) model-specific causal TCBs. To provide a precise foundation, we have introduced a formal definition of the model core along with a model engineering method. To evaluate their feasibility and generality, core-based model engineering has been illustrated by re-engineering three relevant models from different domains. We have shown that even though these models belong to different domains, they share the common model core.

We have evaluated core-based model engineering with respect to its expressive power and the involved modeling costs. We have argued that the model core can express traditional IBAC, RBAC, ABAC models (each in their DAC and MAC variant), including the HRU model, information flow models, as well as trust management models like the RT family. The basis of the model core's expressive power is a modeling framework that is minimal with respect to domain-specific model abstractions. As a consequence, it only remains to specialize the model core. On top of that, the model core provides abstractions that allow specialized models to express policy dynamics.

Besides these advantages, the model core also provides the basis for an causal TCB engineering method. The following section presents this method and discusses its merits.

4 Causal Trusted Computing Bases

If you have a procedure with 10 parameters, you probably missed some.

Alan J. Perlis,
Epigrams on Programming, 1982

TCBs of today's commodity policy-controlled operating systems such as SELinux [165, 166] or SEBSD [240, 246, 247] are characterized by their large functional perimeter. Due to their ambition to provide runtime support for a wide variety of security policies (IBAC, RBAC, ABAC, MLS policies, each in their DAC and MAC variants), the policy runtime environment as well as the enforcement mechanisms are designed in an all-round fashion, rendering them large, complex, and expensive.

This dissertation follows a different approach. The idea is to systematically engineer TCBs by tailoring their policy decision and enforcement environment to support only those security policies that are actually present in the TCBs. A TCB's functional perimeter is then determined by exploiting causal dependencies between security policies and TCB functions. This approach results in policy-specific *causal TCBs* that contain only those functions which are necessary to establish, enforce, and protect their present security policies. Causal TCBs thus set the course for implementations whose size and complexity provide the basis for analyzing and verifying a TCB's correctness and tamper-proofness.

The proposed TCB engineering method is quite similar to core-based model engineering: all causal TCBs share a common TCB component – *a security policy runtime environment (RTE)* – to protect and enforce a wide variety of security policies formalized by core-based security models. Policy-specific causal TCBs are engineered by systematically tailoring the security policy RTE due to causal dependencies between security policies and TCB functions. For this purpose, a formal representation of the causal dependencies is needed, which enables method and tool support for TCB engineering.

The goal of this chapter is to present the TCB engineering approach for causal TCBs. For this purpose, Section 4.1 first presents requirements and prerequisites for designing causal TCBs. In Section 4.2 we then introduce a functional design of causal TCBs along with a general TCB engineering method. Afterwards, Section 4.3 precisely identifies the functional perimeter of a security policy RTE, before Section 4.4 discusses the functional range of policy-specific TCBs. This includes the identification of causal dependencies between security policies and TCB functions that allow for a precise reasoning about the functions to be included in causal TCBs. Section 4.5 concludes with demonstrating causal TCB engineering by means of an example.

4.1 Requirements and Prerequisites

In order to design causal TCBs and develop an engineering method for causal TCBs, we consider three types of requirements: general requirements that arise from the applied TCB engineering approach, design requirements, and architectural prerequisites. The goal of this section is to motivate and discuss these requirements and prerequisites.

To apply the common model core as a specification of the functional requirements of causal TCBs, it needs to be enhanced. Section 4.1.1 motivates and presents these core enhancements. Section 4.1.2 then discusses general requirements and prerequisites that influence both the design and the functional perimeter of causal TCBs. Hardware and architecture prerequisites are considered in Section 4.1.3.

4.1.1 Security Model Core for TCB Engineering

The basic idea of this dissertation is to exploit formal security models to serve as a specification of the functional requirements of causal TCBs. Thus, security models have to be complete with respect to a security policy's responsibilities.

Even though the common model core (Section 3) provides the basis for a uniform formalization of policies, it cannot serve as a specification for the functional requirements of causal TCBs. The reason is that its main purpose so far has been the development of methods and tools for model engineering and analysis. On this account, the model core contains only those operations that modify the policy state (command set C), but lacks operations that merely query the policy state whether their execution is allowed without modifying the state. To implement a policy, however, we need to model the entire policy interface, including state-modifying as well as non-state-modifying operations. As a consequence, the model core needs to consider the latter ones. For example, standard Linux/Unix access control policies protect system calls such as *chmod()* to modify the right assignments of a policy by adding or removing a user's rights. In contrast, *open()* and *read()* [164] are system calls that allow for opening/creating and reading a file descriptor if a user is assigned the required rights.

Definition 4.1 defines the *enhanced model core* by modifying the model components of Definition 3.1 as follows: the set of commands C of the input set $\Sigma = C \times X$ is extended and now contains both state-modifying and non-state-modifying commands. The domain of the state transition function δ is then confined to the set of state-modifying commands.

Definition 4.1 (Enhanced Model Core) The *enhanced model core* is defined as a tuple $(Q, \Sigma, \delta, q_0, E, \lambda)$, combining the model core $(Q, \Sigma, \delta, q_0, E)$ with an output function λ where

- $Q, \Sigma, \delta,\ q_0$, and E are defined in Definition 3.1,
- C is the set of state-modifying and non-state-modifying commands, and
- $\lambda : Q \times \Sigma \rightarrow \{true, false\}$ is the output function.

The output function is defined in analogy to the state transition function δ by Definition 4.2.

Definition 4.2 (Output Function) The *output function* $\lambda : Q \times \Sigma \rightarrow \{true, false\}$ is a Boolean function that implements a positive policy decision iff all of the command-specific conditions are met.

$$\lambda(q, (c, x_p)) \mapsto \begin{cases} true, & \forall cond \in Cond_c : cond(q, x_p) = true \\ false, & \text{otherwise} \end{cases}$$

$q \in Q$ is the current policy state, $c \in C$ is either a non-state-modifying or state-modifying command, $x_p \in X$ are the input parameters according to the formal parameter vector x, and $cond \in Cond_c$ are the conditions of c.

As a result, core-based models are complete with respect to a security policy's responsibilities and may serve as a specification of a TCB's functional requirements. They are now able to explicitly model policy decisions for state-modifying as well as non-state-modifying operations. The latter becomes particularly important whenever the implemented policy needs to export its decisions to other TCB components. For example, the *view()* command of the RBAC policy in Section 3.4.4 is now modeled as follows:

$$\lambda(q, (view, (x_s, x_o))) \mapsto cond_{core}(q, x_s, x_o, view)$$

It should be mentioned that the core enhancements do not influence core-based model engineering since non-state-modifying operations are policy-specific just as state-modifying operations are; they have to be defined during policy engineering. Thus, the enhanced model core can still be applied for model engineering and model analysis. Additionally, it is also qualified for engineering causal TCBs since core-based security models can now serve as complete specifications for the functional requirements of causal TCBs.

4.1.2 Design Requirements

In addition to requirements that arise from the TCB engineering approach, there are requirements that have a direct influence on the design and on the functional perimeter of causal TCBs. Besides, these requirements also confine the application domain of this dissertation. This section motivates and discusses these requirements and thus provides the premise to make design decisions and to identify the required TCB functions (Section 4.2). The main requirements are: Support for

1. Security models in core notation,
2. Multiple security policies,
3. A nonredundant and complete functional perimeter,
4. Substitutionality of security policies, and
5. Engineering methods and tools.

Core Notation To develop a feasible and well-accepted TCB engineering method, a sound formal foundation for security policies is required. Security models in core notation (Section 3) enable such a sound and even uniform formalization of policies, which generally provides the basis for security models to serve as specifications for TCB functions. Additionally, policy specifications in core notation have the following advantages: (i) they are implementation-oriented and follow algorithmic paradigms. Result is only a small semantic gap between formal models and policy implementations, setting the course for an efficient TCB engineering method. (ii) Core-based model engineering allows for modeling a wide variety of access control and information flow control models (Section 3.5.1). Both advantages are significant for this dissertation's goals such that causal TCB engineering exclusively focuses on policies in core notation.

Multi-policy Support For almost two decades, multi-policy systems have gained in importance [33, 114]. According to Hosmer [113, 114] the main reasons are diverse security goals and policies within a single IT system, users belonging to different organizations, or multiple policies for different states of software development [70]. Due to the increasing interest in multi-policy systems, e.g., [33,37,127,142,146,243], causal TCBs must also be able to enforce and protect multiple policies at the same time.

This requirement has a strong impact on the functional perimeter of causal TCBs. It has to contain not only all functions necessary to enforce the entire set of present policies, but also functions that implement the diverse relations between the policies. Here, we distinguish between two main kinds of relations: (i) policies are independent and do not influence each other. In particular, policies have disjoint command and object sets and make independent policy decisions. Causal TCBs thus have to enforce the policies and their decisions independently from each other. (ii) Policies cooperate with each other and make corporate policy decisions. In this case, meta policies like [19,29,36,39,140,168] are required. A meta policy is "a set of rules coordinating the enforcement of multiple security policies“ [113]; they typically specify the order in which policies are composed and enforced, and dissolve conflicts whenever conflicting policy decisions are made [113]. Meta policies thus become a new TCB component. On top of that, TCBs must also provide runtime functions for protecting and enforcing meta policies.

Causal TCBs basically need to be able to support both kinds of policy relations. However, we focus on independent security policies; deriving runtime and enforcement functions for meta policies as well as developing meta policies qualified for causal TCBs is out of the scope of this dissertation.

A Nonredundant and Complete Functional Perimeter Even though the set of functions of a causal TCB is by definition nonredundant and complete, this may not apply to a TCB's implementation. Causal TCBs have to be designed in such a way that the design maintains the functional perimeter's completeness and avoids functional redundancies. This sets the course for establishing redundancy-free but still complete TCB implementations. This requirement becomes particularly important whenever causal TCBs enforce multiple policies of the same type. Policies of the same type can be formalized with the same model; they only differ in their model instances. This implicates that to some extent these policies require the same TCB functions for their enforcement. The TCB design has to consider this by aggregating these functions instead of integrating them multiple times.

Substitutionality of Security Policies Since 2000, IT systems have increasingly become service oriented systems that allow for spontaneous communication and interaction. Thus, today's systems, e.g., [62, 192], are very dynamic applications with spontaneously changing security requirements that frequently require to modify existing application-specific policies and meta policies. Because of this combination of flexibility and dynamics along with the growing demand for security, IT systems need to provide concepts that are able to manage the increasing need of replacing application-specific policies. More precisely, systems must be able to integrate, update, and remove policies ad hoc.

In the context of causal TCBs, modifying a TCB's policies may lead to changing a TCB's functional perimeter. Causal TCBs thus have to be able to reload and unload the relevant TCB functions.

Method and Tool Support The goal of this dissertation is to develop a TCB engineering that can be efficiently and effectively applied to develop causal TCBs for real-world application scenarios. On this account, method and tool support similar to software engineering is required, e.g., design patterns [95], modeling languages and tools [11, 183], or code generators [134, 196, 199]. The objective is to efficiently bridge the gap between formal security models, TCBs, and their implementations. In this context, method and tool support for reusing and generating TCB functions is particularly important since causal TCBs have to be developed individually for each application scenario. The prerequisite is that causal TCBs support such methods by design.

Summary This section has presented basic design requirements and prerequisites for causal TCBs with the objective to confine the application domain of this dissertation. The requirements are mainly motivated by the methodical approach of this dissertation and by recent works on policy-controlled IT systems. The following section introduces design influencing hardware and architecture dependencies.

4.1.3 Hardware and Architecture Dependencies

This dissertation strictly adheres to a top-down design approach that is based on three steps. We first design the functional range of causal TCBs in dependence on their security policies (*functional design*). Afterwards, the TCB functions are embedded in an *architectural design* that requires to meet additional prerequisites such as the reference monitor principles [17, 97, 125]. Finally, the architectural design has to be implemented on a specific implementation platform where implementation details such as programming languages, data types, or privilege levels are of importance (*implementation*). However, a few implementation details already have to be considered during the functional and architectural design of causal TCBs. This section motivates and discusses these details.

Causal TCBs may perform numerous concurrent activities. In order to support these, the functional design of causal TCBs provides an explicit abstraction called `thread`, which models a causally independent activity in a TCB. To what extent these independent and possibly concurrent `threads` may be exploited by a TCB's implementation, i.e., the level of parallelism, remains to be analyzed with respect to specific implementation platforms.

Existing policy-controlled systems such as SELinux [165, 166], SEBSD [240, 246, 247], Solaris Trusted Extensions [79], or Xen [202, 203], were developed by integrating new kernel abstractions into contemporary OS systems. The main advantage of this approach is the compatibility with existing software systems and user applications. On the other hand, this design decision along with the ambition of the new abstractions to be general leads to heterogeneous, large, redundant, distributed, and complex TCB implementations whose functional perimeter cannot be identified precisely. Since this is diametrically opposed to the objective of this dissertation, we decided to develop a novel design.

In addition, the functional TCB design is independent of any operating system (OS) architecture with respect to the standard monolithic and microkernel architecture. For example, let us assume the memory management of monolithic and microkernel-based operating systems. In a monolithic OS the memory management sub-system, including paging and swapping mechanisms, is a large software component that runs in kernel mode. By implementing virtual memory management, it abstracts away a special hardware device, the memory management unit (MMU), while implementing memory protection and easing the implementation of user applications. The memory management sub-system exports a hardware-independent user interface to higher software layers, including functions like *malloc()* and *realloc()* of a Linux/Unix based OS [164], which allow a process to dynamically allocate memory. The MMU implements the translation of physical addresses to virtual memory addresses used by the memory management sub-system to implement its memory management strategies.

In contrast to a monolithic OS, the memory management sub-system of a microkernel-based OS is designed differently in that the only memory management functions running in kernel mode are those that hide the hardware concept of address spaces [161–163]. Consequently, only a very small part of the memory management sub-system runs in privileged kernel mode. Management strategies, including paging and swapping, are usually implemented as servers on top of the microkernel in user mode [51, 150, 161, 162].

From a functional point of view all components are part of the memory management sub-system independent of the mode they run in (analogously thread management and inter-process communication (IPC)). It is thus irrelevant which OS architecture and implementation techniques to consider; only the functionality and the interfaces to higher software layers are significant for this dissertation. On this account, the functional design of causal TCBs is independent from these OS architectures.

Finally, we discuss a hardware requirement that implementation platforms for causal TCBs have to meet – a Trusted Platform Module (TPM) [235]. A TPM is responsible for assuring a system's integrity by building a chain of trust that is rooted in the TPM to protect a system's cryptographic keys and measurement information. Causal TCBs require TPMs to protect their cryptographic credentials, which are needed for protecting and enforcing their policies.

4.2 TCB Design

In due consideration of the above mentioned requirements, the goal of this section is to present a functional TCB design along with a TCB engineering method for causal TCBs. The proposed TCB engineering method is quite similar to core-based model engineering. Analogously to security models, all causal TCBs share a security policy RTE that protects and enforces a wide variety of core-based models. Policy-specific causal TCBs are derived from the functional causal TCB by systematically tailoring the policy RTE due to causal dependencies between policies and TCB functions. A discussion on the plausibility of the functional TCB design and the resultant engineering approach regarding the above mentioned requirements concludes this section.

As shown in Section 3.1, many security models such as [69, 101, 107, 145, 204, 209, 254] share design principles. Based on this observation, we have identified a common model core (Section 3.2) that can be specialized to express a wide variety of existing and novel security policies. Policies that are formalized by models in core notation then share the core's components but may differ in policy-specific specializations of the state, authorization scheme, and extension vector.

We now apply this approach to engineering causal TCBs. The idea is that security policies that share common model components also have identical requirements regarding the functional perimeter of a TCB to enforce them. On the other hand, policy-specific core specialization results in policy-specific requirements and thus in policy-specific TCB functions. On top of that, all policies require a runtime environment ensuring that policies cannot be bypassed and are protected against unauthorized manipulations – typical goals of a reference monitor [17, 97, 125].

From this it follows that causal TCBs contain two different types of functions that are necessary to protect and enforce their present policies:

- Policy-independent functions supporting the model core,
- Policy-dependent functions to support policy-specific core specializations, containing
  - Security model functions, and
  - Security model instance functions.

Before discussing the two types of TCB functions in more detail and deriving a general design of causal TCBs, a common understanding of the term *function* is required. For this reason, we provide a definition of the term function.

Definition 4.3 (Function) A *function* is a mathematical mapping that is described by a domain, a codomain and two additional attributes, called access status and return value, the latter of which is typed and optional.

```
access_status retValue_T functionName(Param1_T name1, Param2_T name2, ...)
```

The `access_status` defines the access level of a function by other TCB components. Each function has one of three possible statuses: `public`, `protected`, or `private`.

The keyword `public` marks a function as interface function to be used by any TCB function and security policy. While `protected` TCB functions are only accessible by policy-independent functions, `private` functions are not accessible by other TCB components; they can only be called by the component that implement them. Moreover, we indicate the types of the input parameters and the return value by *Param_T* or *retValue_T*, e.g., `PPID_T` as type for policy process identifications or `Name_T` as type for names, which are abstract data types. If a function does not have a return value, this is indicated by the keyword `void` instead of a return value type. Having introduced a general understanding of the term TCB function, we now provide details on the two types of TCB functions.

Policy-independent functions are necessary to protect and enforce any security policy considered in this dissertation. They are inherent in any causal TCB and provide a *policy-independent RTE*. On the contrary, policy-dependent functions provide a *policy-dependent RTE*, consisting of functions that depend on both the policies' models and the policies' model instances. Hence, they are individual for any causal TCB. Security model functions are based on the similarities of policies that allow for modeling policies with the same models; they implement the policies' specific models by which they are formalized. Individual characteristics of policies (the authorization scheme) are reflected by model instances and require individual model instance functions to support them.

Based on this differentiation of TCB functions, the design of causal TCBs consists of two main hierarchical layers and two sublayers. Figure 4.1 illustrates the functional design of causal TCBs where the lower layer represents the policy-independent RTE and the higher layer represents the policy-dependent RTE; within the latter there are two functional sublayers that contain security model and model instance functions. Generally it is true that the higher a layer or sublayer is arranged in the TCB design, the higher is the degree of policy dependency of the functions it contains.

The lowest layer contains policy-independent functions necessary to ensure the policies' protection and enforcement, e.g., by isolation and interception mechanisms. This can be

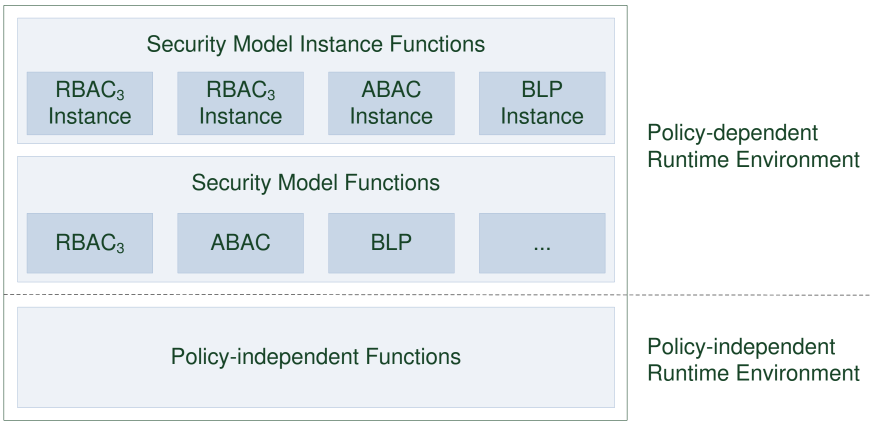
Security Model Instance Functions
RBAC3 Instance
RBAC3 Instance
ABAC Instance
BLP Instance
Security Model Functions
RBAC3
ABAC
BLP
...
Policy-independent Functions
Policy-dependent Runtime Environment
Policy-independent Runtime Environment

Figure 4.1: Functional Design of Causal TCBs

compared to operating systems, where the lowest layer contains basis abstractions and mechanisms to protect and execute the application software running on a OS.

Within the policy-dependent RTE, the lowest sublayer contains functions that support the security models by which the policies have been modeled. For example, let us consider a TCB that among others enforces two dynamic $RBAC_3$ policies and one ABAC policy. This TCB must contain model functions that support these security models. In particular, this sublayer contains the functions that support the $RBAC_3$ model only once (instead of twice) since the $RBAC_3$ policies share similarities that are usually modeled by the same model, requiring the same TCB model functions (Figure 4.1).

In order to support policy-specific functional requirements, the second sublayer contains model instance functions which are required to enforce exclusively those policies that are actually present in a TCB. By this means, the RTE suitable for all policies that can be modeled by the supported model is now tailored to an RTE that only provides support for the present policies.

The result is a functional design for causal TCBs whose functional perimeter is to be tailored to the security policies. Tailoring a causal TCB means to functionally specialize the TCB's policy-dependent RTE by configuring both the security model functions as well as the model instance functions according to the functional requirements of the application-specific policies. The policy-independent RTE is mandatory for all policies in core notation and does not need be tailored.

The design is inspired by microkernel architectures in that they provide a small functional perimeter. In likewise manner, microkernel-based systems provide an RTE for application programs (instead of security policies), which is functionally specialized according to the requirements of the programs. Figure 4.2 presents a functional view of a microkernel-based system. Analogous to causal TCBs, it consists of two main hierarchical layers and two sublayers, where each layer contains a set of functions sharing the same degree of application software dependency. Again, the higher a function is arranged in the layer hierarchy, the more application-depend it is.

The lowest functional layer is represented by the *microkernel* itself, providing a runtime environment that is necessary for executing any possible application software [162, 163]. According to [162], a function is tolerated inside the microkernel if it is mandatory to implement a system's required functionality; basic microkernel functions are thus address spaces, threads, and IPC.

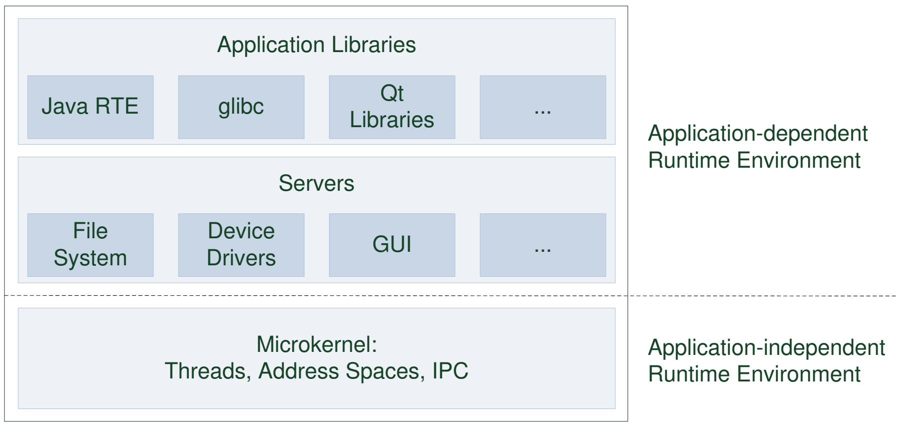
Application Libraries
Java RTE
glibc
Qt Libraries
...
Servers
File System
Device Drivers
GUI
...
Microkernel: Threads, Address Spaces, IPC
Application-dependent Runtime Environment
Application-independent Runtime Environment

Figure 4.2: Design of Microkernel-based IT Systems

Within the application-dependent RTE, the lower sublayer contains typical OS services (called *servers*), e.g., a file system, device drivers, or a windowing system as a component of a graphical user interface (GUI), depending on the application scenario of the system. Considering systems with advanced security requirements for example, the middle layer may contain servers that provide encrypted file system functions like [123] or that provide a windowing manager with special security mechanisms such as [90].

The application-dependent RTE is completed by the topmost layer containing application-specific functions that are typically called *application libraries*. These provide application-specific runtime and library functions such as the *Java Runtime Environment*, *glibc*, or the *Qt* framework.

Analogously to causal TCBs, microkernel-based systems are functionally tailored according to the requirements of the present applications. In contrast to microkernel-based systems where the functional requirements of application software may be arbitrary inspired by any computable application scenario, the functional requirements of causal TCBs are derived from policies that share the same abstractions. More precisely, even though the model core is able to formalize a wide variety of access control and information flow models, the models' syntax is limited and unified. From an engineering point of view, this enables straightforward engineering of causal TCBs with increased efficiency. On top of that, the model-based specification also sets the course for avoiding superfluous TCB functions and functional redundancies.

Having introduced a functional design for causal TCBs and a causal TCB engineering approach, the remainder if this section discusses the plausibility of both with respect to the design requirements of Section 4.1.2.

Core Notation The functional design of causal TCBs follows the same idea as core-based model engineering. For this reason, it optimally supports policies in core notation: analogously to models in core notation, all causal TCBs share a common TCB component – the policy-independent RTE to protect and enforce a wide variety of models in core notation. Causal TCBs are engineered by systematically tailoring the policy-dependent RTE based on causal dependencies between policies and TCB functions.

Focusing on policies in core notation has an impact on the functional perimeter of the policy-independent RTE though (Figure 4.1). In addition to functions that generally protect and enforce the policies, it must also contain functions that support the model core, e.g., by implementing a state automaton, since it is shared by all considered models. On top of that, model functions support policy-specific specializations of the model core, and model instance functions complete the functional perimeter by supporting the instance of the core-based model.

Multi-policy Support To enforce multiple security policies at the same time, causal TCBs have to provide support for policy isolation, policy communication as well as coordination by means of policy composition or meta policies, and scalability with respect to the number of policies. The latter is the only property that can be discussed at the level of TCB design; the remaining properties must be considered at the stage of developing a TCB's architecture and implementation.

Scalability is the ability of causal TCBs to integrate multiple policies without modifying more than the necessary TCB functions. To integrate policies into a causal TCB, the policy-independent RTE does not need to be modified at all; only the policies themselves and policy-specific functions have to be added to the functional perimeter. Here, we have to consider two possibilities. (i) Policies can be modeled by the same model and only differ in their model instances. In this case, the design allows for including the necessary model functions once; only the set of policy-specific model instance functions depends on the number of different policies. (ii) Policies must be described by varying models. The sets of model functions then depend on the number of different policy types; again, the number of policies to be integrated determines the different sets of model instance functions. For example, Figure 4.1 sketches a causal TCB that provides functions for three models and four model instances, hence covering both cases.

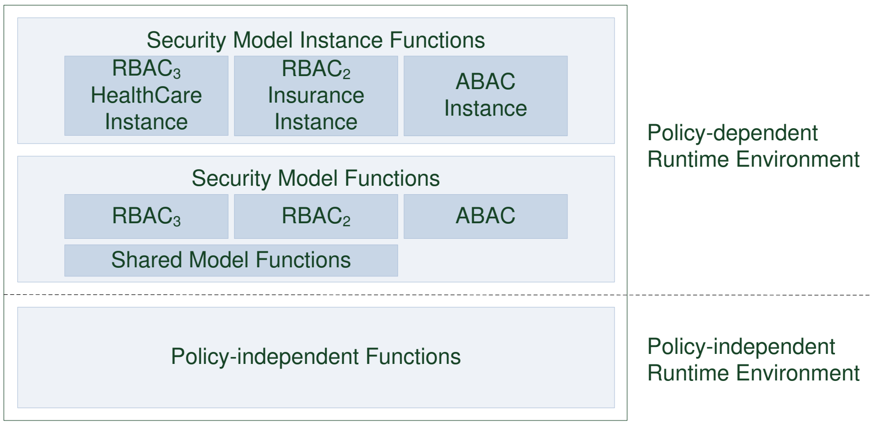
Security Model Instance Functions
$RBAC_3$ HealthCare Instance
$RBAC_2$ Insurance Instance
ABAC Instance
Security Model Functions
$RBAC_3$
$RBAC_2$
ABAC
Shared Model Functions
Policy-independent Functions
Policy-dependent Runtime Environment
Policy-independent Runtime Environment

Figure 4.3: Causal TCB Enforcing an $RBAC_3$, $RBAC_2$, and ABAC Policy

A Nonredundant and Complete Functional Perimeter The causal TCB design along with the TCB engineering approach provides the basis for nonredundant but complete TCB implementations. By tailoring a TCB's functional perimeter to the present security policies, only those model functions and model instance functions that are essential for enforcing the policies are in the TCB. Due to the engineering approach of exploiting causal dependencies between policies and TCB functions, no functions other than causally dependent functions are included. Thus, each set of model functions is redundancy-free and complete by construction with respect to a specific model. The same holds for the set of model instance functions with respect to a specific model instance. However, when integrating more than one policy, e.g., an $RBAC_2$ and an $RBAC_3$ policy, which are formalized by models that share some (but not all) model components, functional redundancy may occur beyond the model function sets.

We deal with this by subsuming the model functions that are shared by these two policies. The model-specific sets of functions then contain only those functions which are specific for a security model. The shared functions also reside in the first sublayer of the policy-dependent RTE, but are arranged below the model-specific function sets so that they have access to the shared functions. In doing so, we avoid functional redundancy when models share more model components than the common model core. Figure 4.3 sketches the design of a causal TCB, tailored to enforce two dynamic RBAC policies – an $RBAC_3$ policy for a healthcare application and an $RBAC_2$ policy for an application for insurance companies.

On top of that, if the policy-independent RTE is designed in such a way that it is nonredundant and complete. Thus, causal TCBs set the basis for nonredundant but complete TCB implementations.

Substitutionality of Security Policies Just like causal TCBs support multiple security policies by design, they also provide support for flexible policies without major changes to their design and functional perimeter. That means, neither the policy-independent RTE nor the policy-depend RTE of other policies is affected. In this dissertation, we distinguish policy dynamics as follows: changes of (i) the model instance, e.g., by adding a new command to the authorization scheme, or (ii) the model, e.g., by adding a new component to the model state. In both cases, the policy-independent RTE does not need to be adapted, only the policy-dependent RTE. Here, the degree of design and functional adaptations depends on the extent of changes. Changes of a model instance require to adapt the set of model instance functions; changes of the security model result in adaptations of model functions, requiring changes to model instance functions, too. Nevertheless, additional model and model instance functions that enforce other policies are not affected by such adaptations.

Method and Tool Support Engineering causal TCBs requires to provide method and tool support that among others allow for reusing TCB functions and generating TCB implementations. This requirement is met by the functional design of causal TCBs as follows. Since the policy-independent RTE is mandatory for any policy in core notation, it can be reused in any application scenario without modifications. In contrast, the policy-dependent RTE is application-specific. Since it is derived from a policy's model and model instance by causal dependencies that can be formally described, this allows for developing an engineering method supported by (semi-) automatable tools, e.g., for generating policy-dependent runtime functions using dedicated compiler-like tools. Additionally, security model functions to support a specific model must be derived only once for any model; they then provide an RTE for any policy instance of the same type. Only the minority of TCB functions, i.e., model instance functions, have to be derived for each individual application scenario.

Summary This section has presented a TCB design that greatly differs from today's general purpose TCBs. The idea is to systematically engineer policy-specific TCBs based on causal dependencies between security policies and TCB functions. Engineering causal TCBs then is similar to engineering core-based security models. All security models in core notation share the common model core that is tailored to formalize specific security policies. Causal TCBs share the policy-independent RTE to protect and enforce all security policies. Here, the policy-dependent RTE is tailored to provide only those functions that are necessary to enforce the security policies that are present in a TCB.

The following sections provide a detailed motivation and discussion of the functions of the policy-independent RTE and the policy-dependent RTE. In doing so, they discuss the functional components of causal TCBs, the abstractions and interfaces they provide as well as their interrelations.

4.3 Policy-independent Runtime Environment

The goal of this section is to motivate and present the TCB functions of the policy-independent RTE. For this purpose, we give a short overview of the functional components of causal TCBs before discussing the policy-independent RTE in more detail.

Figure 4.4 illustrates the functional components of causal TCBs and their interdependencies. Functional components are groups of functions that provide the same basic TCB abstractions. The policy-independent RTE contains ten functional components establishing eight basic TCB abstractions: `security policy process`, `address space`, `thread`, `transaction`, `message`, `gate`, `trusted persistent storage (TPS)`, and `object`[1]. The `AuthenticityTPS` is a special instances of the abstraction `TPS`, which is provided by the component `TPS manager`. The policy-dependent RTE consists of two functional components that provide two basic abstractions (Section 4.4). Each functional component exports at least one interface to be used by security policies and other functional components. The functional components `generic object manager` and `authenticator` are special TCB components; they are the only components that export interface functions beyond the borders of causal TCBs. These functions then define the entrypoints to a TCB. All other interfaces of the functional components are protected and can only be called within the TCB.

We motivate and discuss each functional component of the policy-independent RTE along with its interfaces and interrelations to other TCB components in the following subsections. For this purpose, we present small excerpts of Figure 4.4 focusing on the functional component and its interfaces that is currently discussed.

4.3.1 Security Policy Manager

According to the requirements discussed in Section 4.1.2, causal TCBs have to provide multi-policy support. For addressing this requirement, we refer to a similar problem in operating systems. Operating systems can be considered as runtime systems for arbitrary application software written in various programming languages; and their users have long been familiar with installing and running many applications in parallel on one OS. This is enabled by the OS abstraction ‘process’ and the process manager sub-system that together allow for flexibly executing arbitrary application software by creating, running, and stopping processes. Among others, the process manager includes a process scheduler that allows for running processes concurrently based on a well-defined scheduling strategy. Thus, with respect to application software operating systems already provide well-established abstractions to meet the above mentioned requirements. We now adopt these ideas for causal TCBs.

The `security policy manager` (Figure 4.5) is the central functional component of causal TCBs and can be compared to an OS’s process manager. It provides the abstraction `security`

[1]In order to ease reading, all functional components, abstractions, and functions of causal TCBs will be highlighted by `typewriter font` throughout this dissertation.

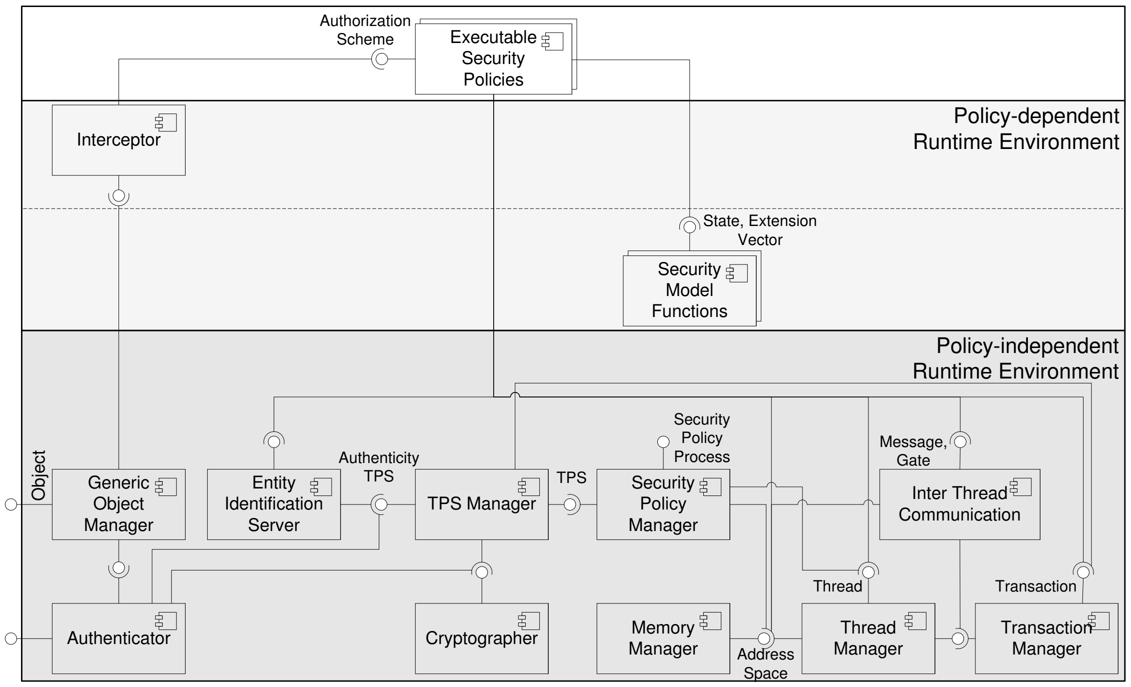

Figure 4.4: Functional Components of Causal TCBs

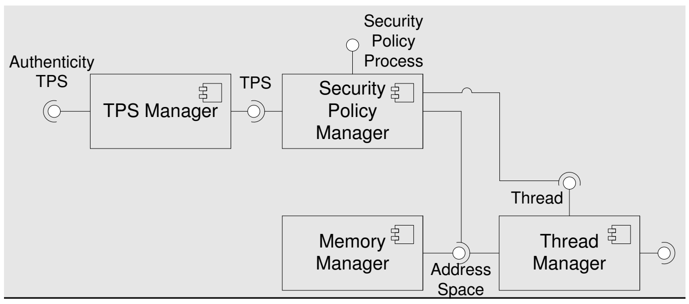

Figure 4.5: Security Policy Manager

`policy process` (also called `policy process`), representing a policy that is executed and enforced by a TCB just as an OS's `process` is an application in execution. Likewise, a `policy process` also has different states. Compared to OS processes though, it may only enter two states as shown in Figure 4.6. When a `policy process` is created, it enters the state *non-enforcing* where it is prepared to be enforced. To enforce a `policy process`, it is started, resulting in that it enters the *enforcing* state. Here, a `policy process` is being executed, which means that it waits for policy requests, makes request-dependent policy decisions, and exports them. A *policy request* is a call of one command of the policy's authorization scheme by some software component outside the TCB; the policy then makes a policy decision, which must be enforced by the TCB. In contrast to an OS, where a `process` is executed only for a limited time before it is terminated or preempted by the scheduler, a `policy process` in general is in the state *enforcing* as long as the policy is not stopped, either to be restarted, e.g., for system restart, or to be destroyed. A `policy process` is destroyed, when its `executable security policy` is modified or removed from a TCB.

In order to manage all `policy processes`, the `security policy manager` implements a

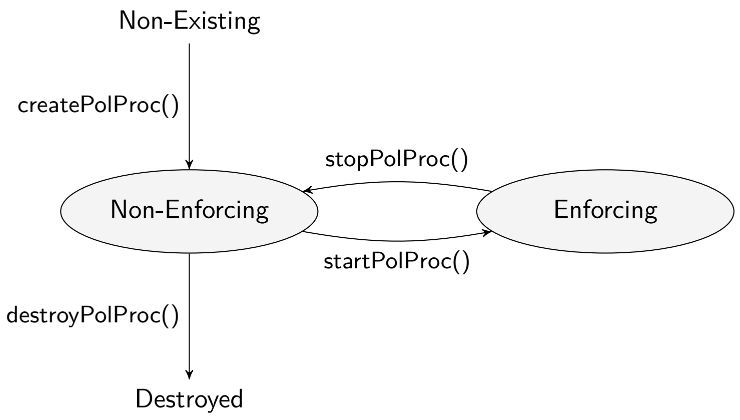

Figure 4.6: Three-State Security Policy Process Model

security policy process control block (PPCB) just as an OS implements a *process control block (PCB)*. Analogously, a PPCB contains identification information of the `policy process` called *policy process identifier (PPID)* and control information to control and coordinate the existing `policy processes`. Control information contains among others information regarding *inter policy process communication (IPPC)*. IPPC enables `policy processes` to communicate with each other in order to make common policy decisions. It thus is a basic mechanism to support multi-policy frameworks such as [38].

A `policy process` is created by the interface function `createPolProc()`. Here, the identifier of the security policy (SPID) to be executed is required, which is created whenever a new policy is integrated in the TCB. When creating a `policy process`, it is also given a unique identifier (PPID) that is only destroyed when the `policy process` is destroyed. That means, a PPID is stored persistently such that a `policy process` has the same identifier after restart. The reason is as follows: security policies can be executed by multiple `policy processes` at the same time. This results in the situation that the policy state must be linked to the executing `policy process` instead of the policy. A persistent PPID is required to permanently map a `policy process` to its policy state, which is also stored in a nonvolatile fashion by a `trusted persistent storage` (Section 4.3.6). In addition, it is required for memory management and thread management.

In order to create a `policy process`, a privileged instance calls `createPolProc()`. Here, the unique PPID is generated, mapped to the `policy process`, and is persistently stored. `startPolProc()` starts a `policy process`, for which its PPCB must be created. A PPCB contains the entire runtime information of a `policy process`; it only exists while a `process` is in the *enforcing* state. As soon as it is stopped by `stopPolProc()`, the PPCB is deleted. In contrast, a PPID is deleted when a `process` is destroyed by `destroyPolProc()`.

```
public PPID_T createPolProc(SPID_T spid)
public void startPolProc(PPID_T ppid)
public void stopPolProc(PPID_T ppid)
public void destroyPolProc(PPID_T ppid)
```

As can be seen, the interface of the `security policy manager` that exports the abstraction `policy process` is not used by other TCB components. The reason is that this interface must be called by a privileged instance – typically known as security administrator – for example via a graphical user interface or a script when booting a TCB. Both are not designed in the context of this dissertation since they do not add to the policy RTE but merely deal with user handling.

The `security policy manager` contains a directory service that allows to search for `policy processes` by their policies; the mapping from the PPID to the identifier of a `policy process` is contained in the `security policy manager`. This also provides the foundation for a multi-policy system where IPPC and thus the location of `policy processes` are required.

```
public List_T findPolProcesses(SPID_T spid)
```

A `policy process` needs to know its PPID whenever it wants to access for instance its `TPS` that stores its state. For this reason, the `security policy manager` provides a function `getPPID()`, returning the persistent identifier of a `policy process`.

```
public PPID_T getPPID()
```

In order to implement the above mentioned interface functions, the `security policy manager` contains the following additional functions, which are `private` and thus not provided by the interface.

```
private void initPPCB(PPID_T ppid, SPID_T spid)
private void deletePPCB(PPID_T ppid)
```

4.3.2 Memory Manager

The `memory manager` is responsible for enforcing tamperproofness of security policies by their isolation. In general, today's isolation mechanisms are based on either software or hardware [9]. Software isolation concepts rely on type-safe programming languages, static compiler-based type-checking mechanisms, and type-checking mechanisms by a run-time system [94, 116, 117]. Hardware isolation via hardware-enforced address space boundaries is an orthogonal approach, which can be found in today's standard operating systems as well as in microkernel architectures [161–163]. While the quality of isolation of both approaches are similar [116], hardware-enforced isolation has the considerable advantage that it requires only a few software functions and an MMU that has long been a standard component of today's computer system. Thus, hardware isolation contributes to a small functional perimeter of TCBs. In contrast, software-enforced isolation depends on the correctness of the compiler-enforced type system, the run-time-based type-checking mechanisms as well as on the overall process of system generation. Additionally, software isolation concepts require all security policies to be expressed by type-safe policy specification languages. This means that the expressive power of the policy specification languages are restricted to enforce type safety. On this account, software isolation concepts are not feasible with respect to the requirements discussed in Section 4.1; we thus decide on hardware-based isolation mechanisms.

In order to establish memory protection and isolation, the `memory manager` protects the hardware that implements virtual memory management. Figure 4.7 shows the functional component `memory manager` and its interrelations to other TCB components. The `memory manager` provides the abstraction `address space`, which is a range of virtual memory addresses. Each `policy process` has its own private `address space`, which has a unique iden-

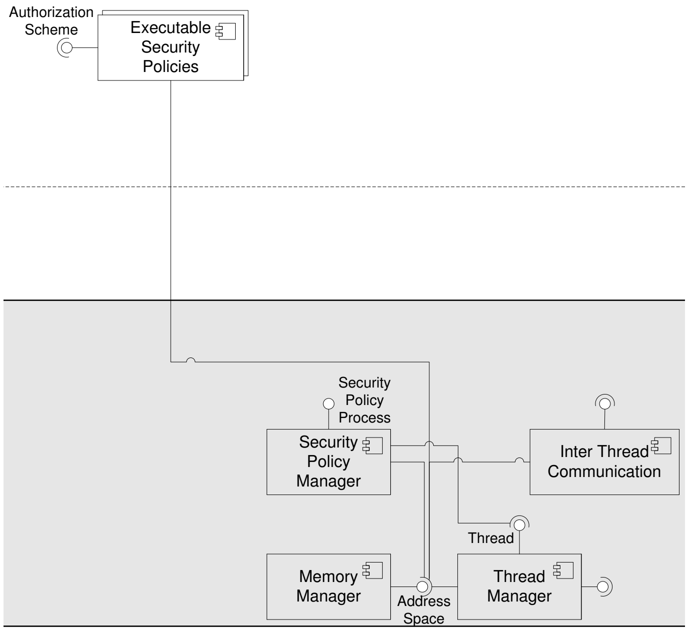

Figure 4.7: Memory Manager

tifier (*address space identifier, ASID*) that is specified by the PPID of the owning `policy process`. The `memory manager` is responsible for managing all existing `address spaces`, including the protection of the `address spaces` from each other. It implements an interface for exporting `address spaces`, which is used by the `security policy manager` for creating and destroying `address spaces` whenever a `policy process` is created or destroyed, by `policy processes`, or by a component called `thread manager`.

When a `policy process` is created by the `security policy manager`, the `memory manager` is called to create an empty `address space` with a unique identifier. Afterwards, a `thread` that has already been created by the `thread manager` is bound to the `address space`, enabling the allocation of memory by mapping the information of the `thread`, e.g., code, stack, and heap, into the `address space`. Accordingly, when a `policy process` is destroyed, its `address space` is also destroyed and its memory is deallocated.

```
public void createAddressSpace(PPID_T owner)
public void destroyAddressSpace(ASID_T asid)
```

A `policy process` can contain more than one thread (Section 4.3.3); all `threads` executing the same `policy process` then run in the policy's `address space`. This enables to parallelize policy decisions in multi-user systems. Since only a `policy process` may know the optimal number of `threads` for its enforcement, `policy processes` are allowed to start and stop threads. Every `thread` then has to be bound to the policy's `address space` by the `memory manager`. `bindThreadToAS()` is provided as an interface function to the `thread manager` so that a `policy process` does not have to deal with the TCB's management functions; it binds a thread to its `address space` by establishing an injective function $f : PPID\_T \rightarrow 2^{TID\_T} \setminus \varnothing$ (*thread identifier, TID*). Accordingly, `removeThreadfromAS()` removes a `thread` from its `address space` when it is terminated. In order to destroy an `address space`, all `threads` running in this `address space` have to be removed from the `address space` and terminated before.

```
protected void bindThreadToAS(ASID_T asid, TID_T threadid)
protected void removeThreadFromAS(ASID_T asid, TID_T threadid)
```

The size of the allocated memory of a `policy process` mainly depends on the policy's dynamic and static components – the protection state and extension vector. These in turn depend on the policy's application scenario, e.g., an ACM with a few thousands of cells of a project-specific documentation application vs. an ACM with billions of cells of an institution's file server [14]. Additionally, due to policy state changes, the required memory size of a security policy may also change during runtime, requiring more memory or less memory. For these reasons, the `memory manager` is not only responsible for isolating `policy processes` from each other, but also for dynamically managing the size of the allocated memory.

```
public Memory_T mem_alloc(Size_T size)
public Memory_T mem_realloc(Memory_T mem, Size_T size)
public void mem_free(Memory_T mem)
```

A `policy process` is allowed to dynamically manage the size of its heap within its `address space` by allocating, reallocating, and releasing memory. This approach equals the dynamic memory management of Unix-based operating systems, which provide among others system calls like *malloc()*, *realloc()*, or *free()* [164]. Within its `address space`, a `policy process` can allocate memory by the function `mem_alloc()`, which returns the allocated memory of type `Memory_T`. `Size_T` is an abstract type for the amount of memory to be allocated. In order to dynamically adapt the size of the allocated memory, a `policy process` must use `mem_realloc()` to expand or reduce the previously allocated memory according to `size`. For deallocating memory for further allocations, the `memory manager` provides `mem_free()` that needs the memory to be deallocated as input parameter.

4.3.3 Thread Manager

A security policy performs numerous concurrent activities: in a policy-controlled OS, applications usually have their individual policies so that many policy decisions may be requested at the same time. Here, we have observed that depending on the enforced policy, the majority of the policy requests does not modify the policy state but only reads it. Additionally, a policy is also responsible for management duties, which are independent of policy requests. Such duties are, for instance, memory management and inter policy process communication. For this reason, the `thread manager` provides the basic abstraction `thread` that enables to execute `policy processes` with concurrent activities.

A `thread` is a sequential, causally independent flow of control within an `address space`. It has a unique *thread identifier (TID)* and an `address space`, which is shared by those `threads` that belong to the same `policy process`. The requirement of causal TCBs to support concurrent activities within a policy along with the general ambition to build a small policy-independent RTE with low complexity, leads to the design of a five-state thread model with the states and state transitions as shown in Figure 4.8; we motivate and discuss the state model in the following.

The state model deviates from standard state models implemented in today's commodity operating systems by lacking the state *new*. The state *new* typically indicates that an OS's execution abstraction – usually a process – is already created but not loaded into main memory due to some limit, based on the number of existing processes or the amount of virtual memory committed to existing processes [226]. For now, we omit this state since TCB's do not have such limits. The reason is twofold: first, the research community has not gained any experience so far about the average number of i) policies running in a policy-controlled system and ii) threads that execute a single policy. Second, as we aim to design a small policy-independent RTE with low complexity, we do not consider any state for which we cannot precisely argue. All other `thread` states are standard states inspired by process models of today's operating systems. In the following, we discuss their relevance and state transitions.

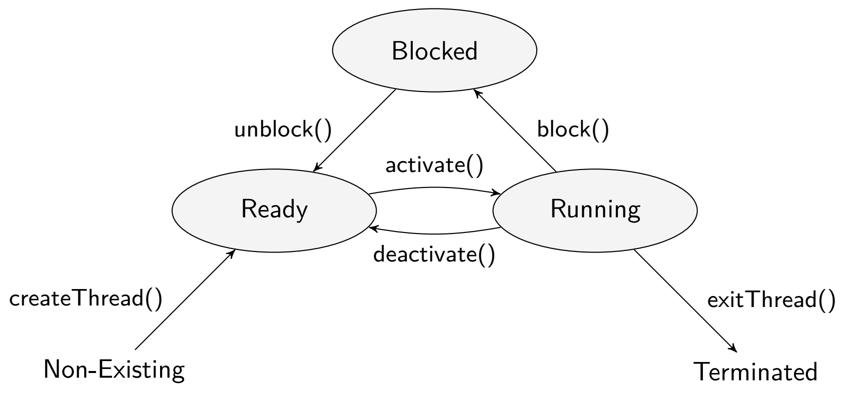

Figure 4.8: Three-State Thread Model

- *Ready*: A `thread` has been created and is now ready to be executed. That means, the `thread` is bound to its `address space` and loaded into memory.
- *Running*: Once a `thread` has been activated by the scheduler of the `thread manager`, it is executed.
- *Blocked*: A `thread` enters the *blocked* state if it requests I/O resources for which it must wait. For example, the functional component `TPS manager` provides nonvolatile storage for a policy's state (Section 4.3.6). This resource is shared by all `threads` of a `policy process`; it thus may happen that a `thread` has to wait before it can access the storage. Moreover, a `thread` enters the *blocked* state when it has to wait for a new policy request. A `thread` returns to the *ready* state when the event for which it has been waiting occurs.

The `thread manager` is responsible for managing and controlling `threads`. Figure 4.9 illustrates the `thread manager` and its interrelations with other functional components of causal TCBs. When a `policy process` is created by the `security policy manager`, the `thread manager` is asked to create a `main thread` for the `policy process`. The `main thread` is a special `thread` of the policy because it is the only `thread` that is not started and terminated by the policy but by the `security policy manager`; all other `threads` are controlled by their `policy process`. As soon as the `main thread` is activated and put in the *running* state, the `policy process` is put in the state *enforcing* where it is initialized and waiting for policy requests. A `thread` must deactivate itself when it can not be executed but is not *blocked.* This is the only state transition that may be triggered by a `thread` itself, and only `threads` implementing transactions (Section 4.3.4) may need to make this state transition. A `thread` is terminated when it has completed its task or is terminated by its `policy process`. The typical reason for termination is that the `policy process` has been destroyed for shutting down the system or for removing the security policy from the TCB. When a `policy process` is destroyed, all `threads` bound to the `policy process` must be terminated.

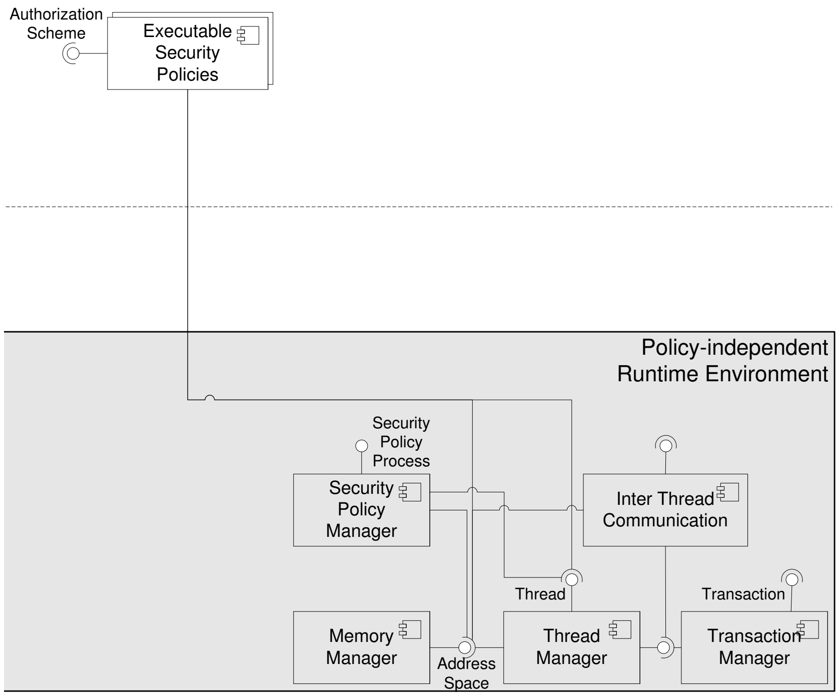

Figure 4.9: Thread Manager

Additionally, the `thread manager` is responsible for scheduling `threads`. For this reason, it maintains a data structure that is called *thread control block*, containing a `thread`'s identification, processor state information, and thread control information. While processor state information contains information about a thread's control and status registers, thread control information consists of scheduling information, including the current thread state, and information about inter thread communication and memory management.

When a `thread` is created by either the `security policy manager` or a `policy process` with the interface function `createThread()`, a unique TID is generated (e.g., for inter thread communication). Afterwards, the `thread` is mapped to the policy's `address space`, allocating the memory for the `thread`, and its thread control block is initialized by `initThCB()`. All `threads` are terminated by the function `exitThread()`. Here, the `thread`'s inter thread communication is stopped, its references are removed, and its memory is freed. Additionally, the thread control block is removed by `deleteThCB()` and its memory is freed.

```
public TID_T createThread(ASID_T asid)
private void initThCB(TID_T tid)
public void exitThread(TID_T tid)
private void deleteThCB(TID_T tid)
```

Whenever the attributes of a `thread` are modified, its thread control block storing these attributes needs to be updated. Among others, these attributes are references to its memory, especially the dynamically allocated memory managed by `policy processes` (Section 4.3.2), and information about inter thread communication. Depending on the attributes, this is either done by the `thread manager` or by the `memory manager` by calling the interface function `updateThCB()`.

```
protected void updateThCB(TID_T tid, Attribute_T attributes)
```

As mentioned above, the `thread manager` is also responsible for scheduling all runnable `threads`. Here, we apply a nonpreemptive first-come-first-serve strategy since it is sufficient and does not add unnecessary complexity. The `thread manager` has an abstract data structure that we call *ready queue*, containing all `threads` in the *ready* state. `Threads` are inserted in the *ready queue* by the function `enqueue()`. `activate()` selects the next `thread` in the *ready queue* for execution; the activated `thread` has to be deleted from the *ready queue* by the function `dequeue()`. If a `thread` deactivates itself by calling `deactivate()`, it releases the CPU and transitions back to the `ready` state. Here it must be inserted at the end of the *ready queue*. Blocking and unblocking of `threads` is implemented by `block()` and `unblock()`. When a `thread` is waiting for an event, the `thread manager` executes `block()` to release the CPU and update the `thread`'s thread control block. `unblock()` is called when an event has occurred; the `thread` then transitions to the *ready* state.

```
private void enqueue(Queue_T queue, TID_T tid)
private void dequeue(Queue_T queue, TID_T tid)
private void activate(TID_T tid)
public void deactivate(TID_T tid)
private void block(TID_T tid)
private void unblock(TID_T tid)
```

4.3.4 Transaction Manager

Whenever concurrently running threads share common resources, the consistency of these resources has to be preserved. In the context of causal TCBs, shared resources are the states of the policies. All `threads` executing one `policy process` use the same state for making policy decisions or changing attribute assignments. Causal TCBs thus have to ensure the states' consistency and isolate all operations performed on policy states, i.e., the commands of the policy's authorization scheme.

This requirement among others has been dealt with by transactional information systems, which guarantee system properties, commonly known as *ACID* properties (atomicity, consistency, isolation, durability) [102,121], that enable concurrent program execution while dealing with failures. By hiding concurrency and failures with the help of a transaction programming interface, clients are easier to develop since they can ignore the complexity of concurrent process execution and failure handling [248].

Atomicity requires that a transaction is completely executed or not at all. That means if a transaction is completely executed, no errors occurred. On the other hand, if a transaction is abnormally terminated, e.g., in case of hardware failure, all the changes of the failed transaction must be undone. Mechanisms to provide atomicity are shadow paging or persistent logging. *Consistency* is defined on the data that a transaction computes; a transaction preserves its consistency when it transitions from a consistent state to another consistent state. This implies that intermediate results are not displayed to the client. There are no self-contained mechanisms to preserve consistency. Instead, a transactional system must be programmed in such a way, e.g., by exploiting constraint declaration statements or triggers, that it eventually reaches a consistent state [248]. If these constraints are violated, a transaction is aborted such that in conjunction with atomicity, consistency is ensured [248]. *Isolation* of transactions ensures that the result of concurrently executed transactions equals the result of sequentially executing these transactions. This means, transactions are isolated from each other when every transaction works on a consistent state that resulted from completely executed transactions. Isolation is usually enforced by pessimistic (conservative) or optimistic (aggressive) concurrency control algorithms. *Durability* of the computed data is preserved when all updates that a transaction has made are guaranteed to survive subsequent software or hardware failures [248]. This implicates that they are stored persistently.

From this it follows that transactions are well suited as semantics for executing policy requests, i.e., calls to the commands of a policy's authorization scheme. However, they come with great efforts in terms of complexity and thus TCB functions. On the other hand, a high degree of parallelism provides the basis for applying causal TCBs in real-world systems where multiple users require applications with high performance. Consequently, the goal must be a compromise between the required high degree of parallelism and the efforts involved. Our approach is to support only those ACID properties that are indispensable for causal TCBs by implementing a *lightweight transaction management*. On this account, we discuss the significance of each property for causal TCBs in the following.

The main responsibility of causal TCBs is to ensure that the present security policies are correctly enforced. This includes that (i) policy decisions are always based on a consistent policy state and (ii) the result of executing concurrent policy requests must be identical to the result of sequentially executing the same requests. Hence, consistency of the policy states and isolation of the policy requests are indispensable properties that need to be preserved by causal TCBs.

There is no requirement that causal TCBs need to deal with hardware or system failures; in this case, a policy may indeed reach an inconsistent state. However, to correctly enforce its policies, a TCB must still ensure the consistency of policy states even though system failures occur. This means, atomicity is an important property of causal TCBs that contributes to preserve the consistency of all states. Additionally, to correctly enforce policies beyond system restart, the states of all policies have to be stored in a nonvolatile and consistent fashion. The policy-independent RTE supports this by the abstraction `trusted persistent storage`. However, the policies themselves are responsible for storing their states, i.e., they define the degree of durability they want to preserve.

To guarantee the properties atomicity, consistency, and isolation (ACI), and to support a policy in its management tasks, the `transaction manager` provides a transaction programming interface that implements a lightweight transaction management. The programming interface is implemented via *transactional memory* [103, 111, 149], which guarantees the ACI properties for concurrent activities within shared memory by design. `Threads` that belong to a `policy process` then communicate with each other via objects in main memory by applying transactional semantics. This allows for a high degree of parallelism and an easy programming interface, since a policy only needs to identify its critical sections. In case the hardware of a platform-specific TCB implementation does not contain transactional memory, the programming interface can be implemented in software. Besides, the algorithms to implement the ACI properties, e.g., optimistic or pessimistic concurrency control algorithms, depend on a TCB's specific application scenario. Thus, they have to be designed during the implementation of a specific causal TCB.

Independent of a TCB's implementation, any `transaction` has two states: *running* and *committed* (Figure 4.10). We have omitted a state *aborted* as we aim to design a small policy-independent RTE with low complexity and without dispensable states. The reason for the latter is twofold. First, a policy does not need to abort a `transaction`. Second, the `transaction manager` can internally enforce that `transactions` are restarted and hence eventually created and executed. By this means, the policy does not have to restart a `transaction`; this is automatically done by the `transaction manager`.

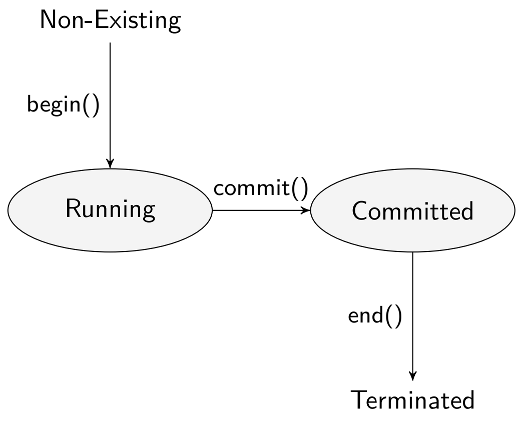

Figure 4.10: States and State Transitions of a Transaction

`Transactions` are implemented by `threads`. A `thread` must be in the *running* state (Figure 4.8) to execute a `transaction`. If a `transaction` cannot be created, its `thread` will transition back to the *ready* state by calling `deactivate()`. In case a `transaction` is terminated, its `thread` may be terminated, too. However, a `thread` may also enter the *blocked* state and execute another `transaction` afterwards.

The `transaction manager` provides the following transaction programming interface. A `transaction` is started by `begin()`, where the `transaction manager` created a `transaction`, prepares it for execution, and finally returns a unique *transaction identifier (TAID)*. A `transaction` is committed by calling `commit()`. To terminate a `transaction` afterwards, a policy calls `end()`, where the `transaction manager` not only destroys the `transaction` but also its identifier.

```
public TAID_T begin(List_T locks)
public void commit(TAID_T taid)
public void end(TAID_T taid)
```

The transaction programming interface guarantees atomicity, consistency, and isolation of all policy states and policy requests. Though it is basically applied by policies, it can also be used by any other TCB component that computes the policy states. As shown in Figure 4.12, such a component is the `trusted persistent storage (TPS) manager`, which is responsible for storing policy states in a trusted and nonvolatile fashion. Its motivation and details are given in Section 4.3.6.

4.3.5 Inter Thread Communication

In order to realize the above mentioned design concepts, `threads` have to communicate with each other. Fur this purpose, the functional component `inter thread communication (ITC)` provides abstractions and functions that enable thread communication beyond `address space` boundaries. Here, we distinguish two application scenarios:

- `Security policy processes` are isolated from the policy-independent RTE by different `address spaces`. Thus, `threads` must be able to switch their context, in order to use the services of the RTE.
- `Security policy processes` are also isolated from each other by different `address spaces`. In order to lay the foundation for *inter policy process communication*, e.g., for implementing meta policies, the `threads` of various `policy processes` must be able to communicate with each other.

The first scenario is equivalent to making system calls in operating systems. Here, a process running in user mode must make a context switch in order to be able to execute the operation in the address space of the kernel. Analogously, whenever a `policy process` uses a function of the policy-independent RTE, it must switch from its own `address space` to the `address space` of the RTE. For this reason, the `ITC` component provides the function `trap()`, which is called whenever a function from the RTE is called by a `policy process`. With the help of the `thread manager` (`updateThCB()`) and the `memory manager` (`bindThreadToAS()`), the `policy process` is able to jump to the address of the called RTE function in the `address space` of the policy-independent RTE, to execute this function, and to jump back to its own context. For this purpose, the RTE function as well as its parameters are input parameters of `trap()`, from which the jump address can be retrieved. Besides, `trap()` must save the return address so that the `thread` of the `policy process` can return to its context.

```
protected void trap(Function_T function, List_T paramList)
```

Figure 4.11 shows the `ITC` component and its interface to the security policies. For inter policy communication, the `policy processes` directly use the interface of the `ITC` component. For context switches, the `ITC` interface is called by the other TCB components; for clarity, we refrain from illustrating the usage of this interface by all other TCB components that provide services to `policy processes`. Besides that, the `ITC` component must be able to communicate with the `thread manager` and the `memory manager` to actually implement the communication between `threads`.

The second use case can be compared to message-based client/server communication where meta policies are clients that request partial policy decisions from the participating security policies (servers) and combine them to a common policy decision. Based on these considerations, the basic abstractions are synchronous *send* and *receive* `messages` that are sent between `threads` of different `address spaces`.

A `policy process` does usually not know from which `thread` (representing a client) to receive `messages`. Consequently, it must be able to receive messages from any `thread` belonging to another `policy process`. For this reason, the communication is indirect, and similar to [136, 237], it uses an abstraction `gate` that is bound to a `thread` as mediator. Due to confidentiality, there is no central register for `threads` belonging to a specific `policy process`. That means, whenever a `policy process` wants to communicate to another `policy process`, there is no means of identifying the policy's `threads`. `Gates` are bound to `threads`

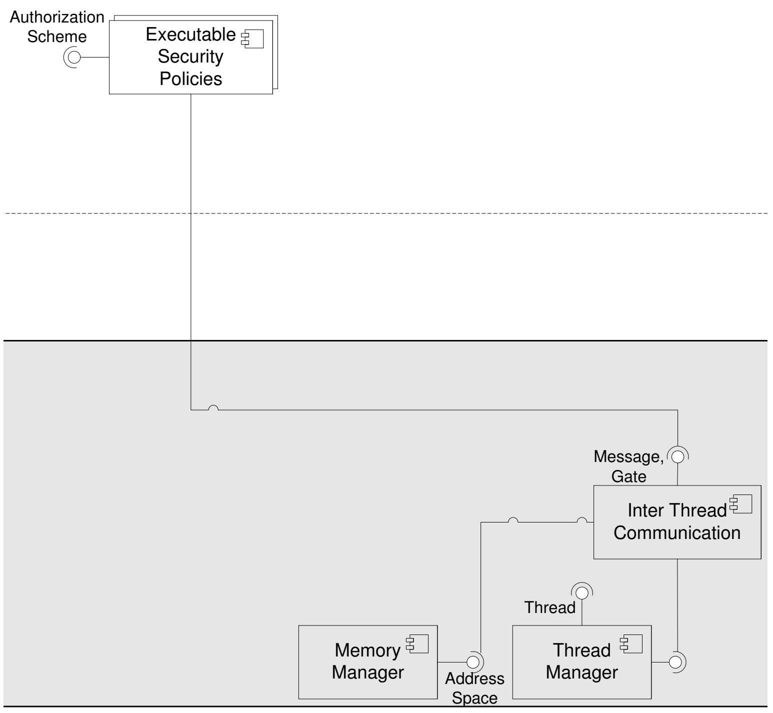

Figure 4.11: Inter Thread Communication

by a mapping $f : TID\_T \rightarrow 2^{GATE\_T}$ that is managed by the `ITC` component. `Gates` can then be found by calling `findGatesForPolProc()`, which returns a list of all `gates` that are bound to the `threads` of the `policy process`.

Once a `gate` of the partnering `policy process` is found and the policy's own `threads` have created their `gates` and have bound them by `createGate()` and `bind()`, the `policy processes` can communicate via sending and receiving messages using `send()` and `receive()`.

```
public void send(Gate_T receiver, Msg_T msg)
public Msg_T receive(Gate_T sender)
public Gate_T createGate(Attributes_T attributes)
public void destroyGate(Gate_T gate)
public void bind(TID_T tid, Gate_T gate)
public List_T findGatesForPolProc(PPID_T ppid)
```

As mentioned in Section 4.1.2, causal TCBs need to support two kinds of policy relationships: independent and cooperating security policies. By establishing `gates`, security policies are provided with an abstraction that is generally able to implement policy-specific communication models using message filters. This may be particularly important for supporting cooperating security policies. For example, in order to make a common policy decision, a meta policy requests policy decisions from all participating security policies. However, these policies may only be called by a specific meta policy but not by others. `Gates` can then implement, for example, message filters in order to prevent unauthorized messaging. By this means, an additional level of communication, which we call *inter policy process commu-*

nication can be realized on top of `ITC` to implement policy-specific communication models for cooperating `policy processes`. It remains future work to research which types of co-operations between security policies may exist and thus which communication models are required.

After having discussed the functional components responsible for security policy execution, the remaining TCB components (`TPS manager`, `cryptographer`, `entity identification server`, `generic object manager`, and `authenticator`) deal with establishing integrity and confidentiality of the policy states while guaranteeing authenticity of all entities controlled by the policies.

The `TPS manager` is responsible for storing all policy states in a trusted and nonvolatile fashion. At the same time, it provides the basis for establishing authenticity of all policy entities by storing their identifiers as well as authentication information. The `cryptographer` provides the required cryptographic mechanisms for trusted storage. Entities are abstract representations of the objects managed by an OS, whose authenticity needs to be preserved, too. This is realized by the cooperation of three TCB components: `entity identification server`, `generic object manager`, and `authenticator`. The `entity identification server` centrally manages the identifiers of all entities, which must be bound to the OS objects. This is the responsibility of the `generic object manager` by establishing a logic interconnection between entities and their implementation as runtime or persistent OS objects. For this purpose, the `generic object manager` contains all functions that are relevant for policy enforcement independent of the object type. The implementation of a TCB then contains a collection of `specific objects managers`, which inherit the functions of the `generic object manager` but implement them for their specific objects. The `authenticator` can be considered a implementation-independent `specific object manager` that is responsible for establishing the authenticity of all objects representing human users. We motivate the remaining functional components and give details on each in the following subsections.

4.3.6 Trusted Persistent Storage Manager

TCBs are responsible for correctly enforcing their security policies at any time. While the `memory manager` protects the policies during runtime via isolated `address spaces`, the `trusted persistent storage manager` is responsible for isolating the persistent memory of policies, storing their protection state.

In order to store the protection states of security policies, causal TCBs have to ensure that the states' integrity and confidentiality are preserved. Integrity, i.e., the property of information to be protected against unauthorized modification, ensures that no other security policy or system component outside the TCB is able to manipulate the policy state, e.g., with the goal to grant some subject specific rights for an object that the subject would not have otherwise. Without maintaining the integrity of the policy state, causal TCBs cannot guarantee that their policies are correctly enforced with respect to their specification. In contrast, confidentiality (the property of information to be available only to authorized users) contributes to the correct enforcement of security policies rather indirectly. Policy states contain sensitive information about subjects and objects, e.g., subject/object attributes like age, salary, rights, or any other attribute used for policy decisions. If any other system component or policy may gain this kind of information, it does not have an immediate impact on the policy enforcement. Nevertheless, an adversary gets fundamental insights in a policy, which can then be used to accomplish well-directed attacks applying social engineering methods. Hence, the security policy will be undermined. Of course, guarding IT systems against social engineering attacks is not possible; we can merely raise the bar by preserving the confidentiality of policy states.

Responsible for storing the policies' states and maintaining their integrity and confidentiality is the TCB abstraction `trusted persistent storage (TPS)`. Every policy has its own `TPS` where it can save its protection state on trusted, nonvolatile storage. A `TPS` is attached to a `policy process`; only `threads` that belong to the `policy process`, i.e., that reside in the policy's `address space` (Section 4.3.2), can access the policy's `TPS`. Since the policies' states may greatly differ in structure and size, a `TPS` does not provide a means of logical organization; it is the policies responsibility to logically structure its data. Any `TPS` has four attributes: a unique identifier specified by the owner of the `TPS` (the PPID of the `policy process`), its size, a set of references to blocks of a data medium that contain the policy data, and the current hash of the policy data stored in the `TPS`.

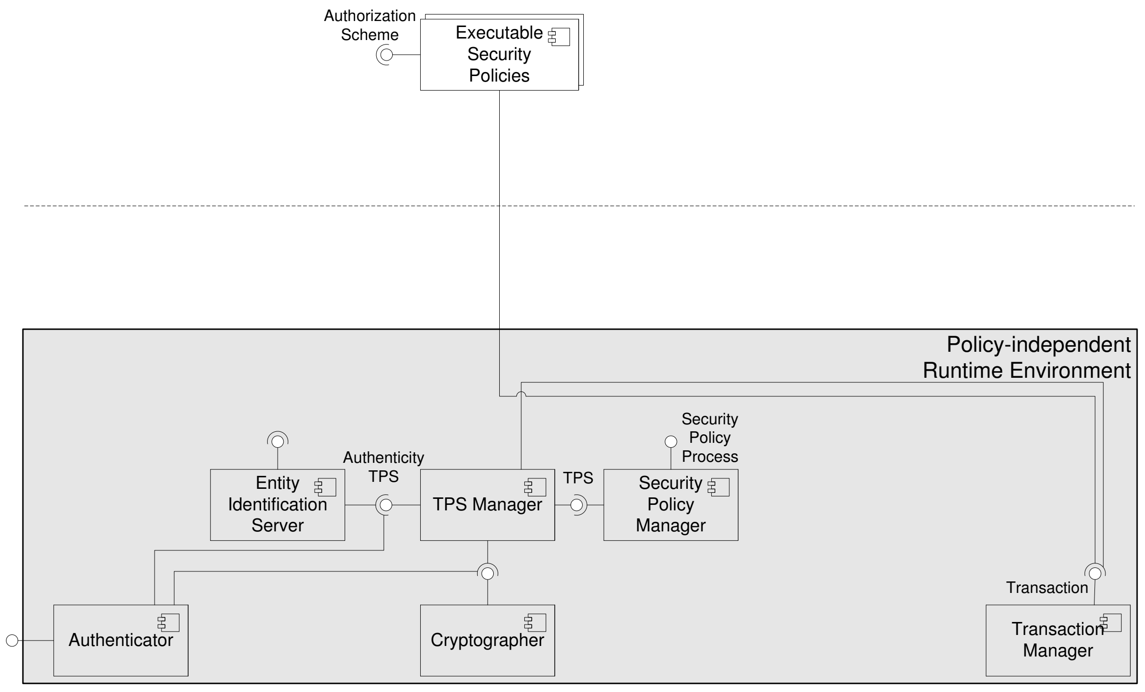

Figure 4.12: Trusted Persistent Storage Manager

The `TPS` abstraction is provided by the `TPS manager`, implementing a nonhierarchical storage system whose storage units are well isolated by cryptographic mechanisms. Figure 4.12 shows the `TPS manager` and its interrelations to other TCB components. The `TPS Manager` manages the `TPS`' of all security policies integrated in a causal TCB by providing a collection of functions that can be performed by the policies on their `TPS`.

```
public void open()

public Data_T read(Size_T size)

public void write(Data_T data, Size_T size)


public void close()
```

Before a `policy process` is able to perform a function on its state, its `TPS` has to be opened.

Here, it is important that the `TPS manager` ensures that a `TPS` can only be opened by its `policy process`. Since the PPIDs of the `policy process` are persistent, the mapping from the owning `policy Process` to its `TPS` well defined at any time and can easily be enforced by the `TPS Manager`. Once a `TPS` is open, a `policy process` is allowed to perform functions on its `TPS`. The function `read()` enables to read all or a portion of the policy data in the `TPS`, depending on the input parameter `size` that indicates the amount of data to be read. Policy data can be stored persistently by using `write()`, whose input parameters are the `data` and the `size` of the data to be written. A `policy process` can write policy data either by updating the existing data or adding new data. Since `read()` and `write()` are performed directly on a policy state, the `TPS manager` applies the transaction programming interface provided by the `transaction manager` (Section 4.3.4), to ensure a state's consistency and the isolation of both functions. When a `policy process` is terminated, its `TPS` has to be closed. `Threads` are then no longer allowed to perform functions on the `TPS`.

As already mentioned, the data stored in a `TPS` is protected by cryptographic mechanisms; it is encrypted and hashed for maintaining its confidentiality and integrity. We refrain from using digital signatures, since they do not add to the degree of integrity here. The reason is that the hashes of the `TPS` will never be exposed to any other security policy or to the outside of the TCB so that they cannot be modified unauthorizedly. That means, the hashes are encrypted by the `TPS Manager` before they are persistently stored, and the hash attribute of the `TPS` abstraction resides in the protected `address space` of the owning `policy process`. Moreover, asymmetric encryption is much more expensive in terms of complexity, implementation efforts, and performance. Thus, we recommend the usage of symmetric encryption protocols for encryption, because on the one hand, asymmetric encryption is not strictly necessary; on the other hand, redundant encryption functions for asymmetric and symmetric keys can be avoided.

Applying cryptographic mechanisms involves that every time policy data is written to a `TPS`, it has to be encrypted and the hash attribute of the `TPS` has to be updated. Moreover, the hash is stored persistently and encrypted by the `TPS Manager` to be used after system restart. Encrypting and hashing the contents of a `TPS` every time is feasible since policy requests that modify a policy state are usually rare[2]. If a `TPS` is read, the data has to be decrypted and the hash has to be checked. In case the result is negative, i.e., when a policy's data has been modified unauthorizedly, a `policy process` can no longer trust the stored data to make policy decisions. Consequently, the enforcement of the policy must be stopped. A privileged instance like an administrator then has to manually analyze the policy state and confirm the state's correctness. This task may be very burdensome depending on a policy's complexity and the size of its state. A policy does not need to encrypt, decrypt, or hash the content of its `TPS` itself. All cryptographic functions are initiated by the `TPS manager` by calling the interface functions of the functional component `cryptographer` (Section 4.3.7).

Furthermore, the `TPS manager` provides a set of functions for managing a policy's `TPS`. This includes the creation of a new `TPS` when a new policy is integrated into a causal TCB, and the destruction of a `TPS` when removing a policy from the TCB. A new `TPS` must be created before the `policy process` of the policy is started the first time; this can be accomplished when the program code of the executable security is integrated into the TCB, e.g., by an administrator. Here, the `size` of the `TPS` is an input parameter, since the size of the states are policy-dependent and usually very different. On top of that, the required size of a `TPS`

[2]Note that for performance reasons, reading and writing a policy's `TPS` can be independent of policy requests; it suffices to synchronize the `TPS` with the virtual state in regular intervals.

can change according to the requirements of the states of the policies. Thus, in analogy to memory management (Section 4.3.6), a policy is able to increase (`s_alloc()`) and decrease (`s_dealloc()`) the size of its `TPS` as indicated by the input parameter `size`. Whenever the size of the `TPS` or its hash has been modified, its attributes must be updated by `updateTPS()`.

```
protected TPS_T createTPS(PPID_T owner, Size_T size)
protected void destroy(TPS_T tps)
public void s_alloc(Size_T size)
public void s_realloc(TPS_T tps, Size_T size)
private void updateTPS(TPS_T tps, Attribute_T attributes)
```

Besides the storages of the policies, the `TPS Manager` is responsible for two specific `TPS`' that store the data of a causal TCB itself: `CryptographicTPS` and `AuthenticityTPS`. Analogous to all other storages, they are also encrypted. In contrast to the policies' storages where the policies are responsible for their logical organization, the organization of the `CryptographicTPS` and `AuthenticityTPS` is managed by the `TPS manager`. Both must be initialized when setting up a causal TCB; they must not be destroyed.

The `CryptographicTPS` holds the hashes of all `TPS`' to ensure their long-term existence. The hashes are mapped to the identifiers of the owning `policy processes`. Since the `CryptographicTPS` and `AuthenticityTPS` do not belong to a `policy process`, their hashes are mapped to virtual PPIDs. In order to read and write the hashes, the `TPS manager` implements two functions which can only be accessed by itself. `retrievHash()` retrieves the hash of a `TPS` by the PPID of its owning `policy process`. The `TPS manager` stores the current hash of a `TPS` by `saveHash()`; this function also saves an additional hash of a new `TPS`. When a security policy is removed, its `TPS` is destroyed along with its hash in the `CryptographicTPS`. The latter is done by `destroyHash()`.

```
private Hash_T retrieveHash(PPID_T owner)
private void updateHash(PPID_T owner, Hash_T hash)
private void destroyHash(PPID_T owner)
```

The `AuthenticityTPS` saves all data that is required by the functional components `authenticator` and `entity identification server`. Both are responsible for ensuring the authenticity of the subject entities protected by the policies. As such they require to permanently store entity identifications and credentials for user authentication. For retrieving and saving the information of the `AuthenticityTPS`, the `TPS manager` provides four interface functions. `retrieveEntityIDs()` allows the `entity identification server` to gain access to all assigned entity identifiers, e.g., after system restart; `saveEntityIDs()` saves the entity identifiers. The `authenticator` can retrieve an credential for a specific subject entity by `retrieveCredential()`; a credential for a subject entity is saved by `saveCredential()`. For more details, please refer to Sections 4.3.8 and 4.3.10.

```
protected EntityList_T retrieveEntityIDs()
protected void saveEntityIDs(EntityList_T entities)
protected CredentialList_T retrieveCredential(EID_T user)
protected void saveCredential(EID_T user, Credential_T cred)
```

4.3.7 Cryptographer

A causal TCB requires cryptographic mechanisms to ensure the confidentiality and integrity of permanently stored data. On this account, the `cryptographer` provides a collection of cryptographic functions; their precise configuration, e.g., encryption algorithm, key length, or hash function, depends on the implementation. Figure 4.13 shows the `cryptographer` along with its interrelations to the `authenticator` and `TPS manager`.

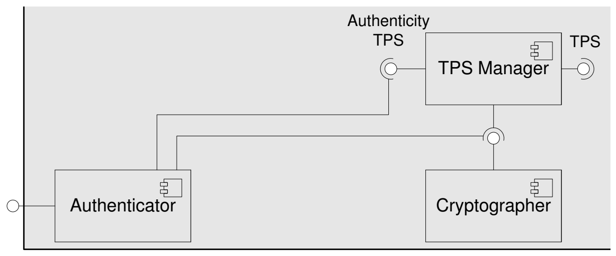

Figure 4.13: Cryptographer

As already mentioned, the `cryptographer` implements symmetric encryption. Consequently, each causal TCB must have its own symmetric key for encrypting their storages; in the following we refer to this key as *TCB key*. The TCB key is generated by the `cryptographer` via `generateKey()`. Naturally, it is of utmost importance to preserve the confidentiality of the TCB key. This is done by the *storage root key (SRK)* of the TPM by integrating it in the key hierarchy managed by the TPM. This means the SRK encrypts the TCB key before it is saved on untrusted storage. The `cryptographer` can gain access to the key by the function `loadKey()` that locates the TCB key and calls the TPM to decrypt it with the SRK.

```
private Key_T generateKey()
private void saveKey(Key_T key)
private Key_T loadKey()
```

The `cryptographer` provides the function `encrypt()`, which produces cyphered data depending on the `plain data` to be inputed. The function `decrypt()` deciphers data, respectively, requiring the `ciphered data` as input parameter. Since there is only one key for data encryption, i.e., the *TCB key*, the key does not need to be an input parameter.

```
protected Cypher_T encrypt(Plain_T plainData)
protected Plain_T decrypt(Cypher_T cypherData)
```

On top of that, the `cryptographer` provides two functions for computing hashes and performing checks on them. The function `hash()` produces a hash code from the input parameter `plainData`; it is necessary that `hash()` is implemented as a one-way and collision resistant hash function. Once the RTE is able to produce hashes, they need to be compared to measure the integrity of the hashed data by `checkHash()`. The first input parameter is the newly created hash that needs to be compared with the original hash of some data (second input parameter), e.g., the hash attribute of an `TPS`. `checkHash()` returns `true` if the hashes are equal; it returns `false` otherwise.

```
protected Hash_T hash(Plain_T plainData)
protected Boolean_T checkHash(Hash_T hashTocheck, Hash_T originalHash)
```

All cryptographic functions are `protected` interface functions to be used by the TCB components `authenticator` and `TPS manager`. Security policies do not have to take care that their states and user authentication data is protected; this is a service of the policy-independent RTE provided to all security policies.

4.3.8 Entity Identification Server

One corner pillar of establishing and maintaining the authenticity of the policies' entities is their unambiguous identification. On this account, globally unique identifiers are required by which any entity can be identified at any time, independent of its type, the represented OS abstraction, or the policies by which it is protected. The `entity identification server` is responsible for centrally managing the identifiers of the entities (*entity identifier, EID*) in order to enforce their uniqueness. Conceptually, EIDs can be realized in two ways: *nonreusable* or *reusable* EIDs. If EIDs are nonreusable, there will be no EID that is reused by a new entity when its old entity is destroyed even after system restart. By this means, it is very simple to enforce unique EIDs; however, the identifier space, no matter how large it is, will overflow eventually. Hence, nonrepeating identifiers are not feasible. In contrast, EIDs that will be reused by new entities right after their old entities are destroyed hardly pose the problem of overflown identifier spaces; though the efforts for guaranteeing that entities with reused EIDs do not gain the permissions of the former entities increase enormously. Since both approaches are suboptimal, we choose an hybrid approach where EIDs may be reused on specific conditions: after system restart and when the identifier space is completely traversed. The advantage is that the identifier space cannot overflow when a feasible range is chosen, while the effort for guaranteeing that entities with a reused EIDs do not gain old permissions is now moderate.

In order to give security policies the opportunity to create or remove entities, the `entity server` provides two interface functions for requesting new and destroying old EIDs. Whenever a policy is provided an EID by the `entity server`, it can depend on the uniqueness of the identifier and can hence assign credentials, e.g., permissions, roles, or attributes. On the other hand, when an EID is destroyed by a policy, it's the policy's responsibility to ensure that all references to this EID are deleted. Figure 4.14 shows the functional component `entity identification server` and its interrelations.

```
public EID_T createEntityID()
public void destroyEntityID(EID_T entityID)
```

As already mentioned, an EID is bound to an OS object by the `generic object manager` as long as the object and thus the entity exists. This requires that EIDs of persistent objects are also bound persistently. However, the `entity identification server` also needs to know the persistent identifiers after system restart for generating new EIDs. For this reason, the persistent EIDs are also centrally stored in the `AuthenticityTPS`, which is provided by the `TPS manager`. By using the interface function `retrieveEntityIDs()`, the

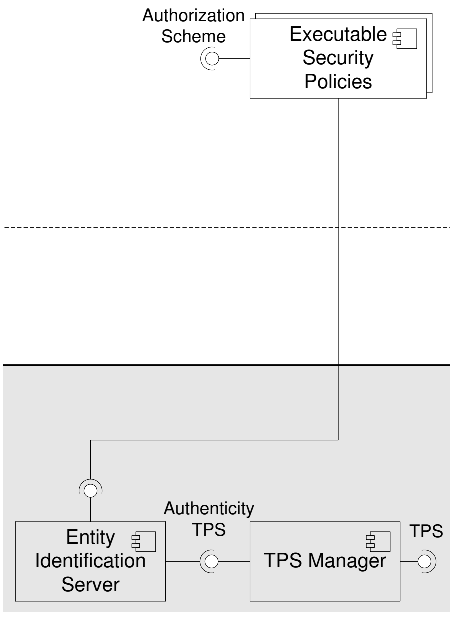

Figure 4.14: Entity Identification Server

`entity identification server` can load the EIDs from the `AuthenticityTPS` that are currently bound to persistent objects. The function `saveEntityIDs()` enables the `entity identification server` to store the EIDs that are assigned to objects.

4.3.9 Generic Object Manager

The second corner pillar of establishing and maintaining the authenticity of the policy entities is to correctly and irrevocably bind the EIDs managed by the `entity identification sever` to their object representation within an OS. This is the responsibility of the `generic object manager`, which establishes a logic interconnection between policy entities and their implementation as runtime or persistent objects in an OS. For this purpose, the `generic object manager` provides the abstraction `object` and a collection of functions for their management. Figure 4.15 shows the `generic object manager` along with its interfaces. As shown, the `generic object manager` also provides an interface to the outside of the TCB. This interface presents one of two entrypoints into the TCB for the policy-controlled OS. Every time the `generic object manager` receives a request, it calls the `interceptor` which then calls the security policy for making a policy decision.

Security policies of policy-controlled operating systems are responsible for protecting any object managed by an OS. Since these objects may greatly differ in their attributes, management operations, and operations to be performed on them, operating systems typically implement an object manager per object type (OS abstraction) or group of types. For example, the file system is the object manager for the object types file and directory, which have to be stored permanently and which can be created, copied, read, written to, or linked by possibly many users at the same time. On the other hand, processes are runtime objects managed by the process manager, which are created, executed, scheduled, or terminated. Binding the EIDs to objects of different types and managing them thus requires different implementation mechanisms. However, at the abstraction level of TCB functions there is no difference between objects of different types, e.g., on this abstraction level managing files,

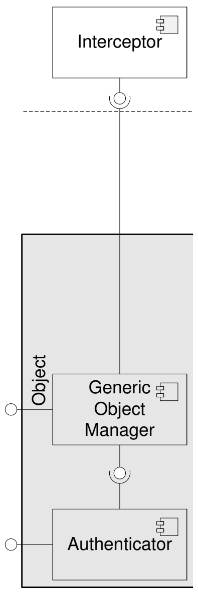

Figure 4.15: Generic Object Manager

sockets, processes, threads, or pipes does not make a difference because for all holds, that an EID has to be correctly and irrevocably bound to them such that they can be clearly identified at any time. On this account, causal TCBs contain an object type-independent `generic object manager`, providing object type-independent functions for object management. For implementing a policy-specific TCB, type-specific object managers must be derived, inheriting the functions of the `generic object manager`, which then have to be implemented according to the specific object type.

Inspired by object orientated software design, the `generic object manager` provides the generic TCB abstraction `object`, having abstract attributes that are shared by all object types of an OS. When deriving type-specific object managers for a TCB's implementation, their specific object types have to be derived from the `object` by implementing the abstract attributes. On top of that, the `generic object manager` provides a collection of abstract functions. Analogous to the abstract attributes, any `specific object manager` has to implement the abstract functions depending on its specific object type. The fundamental requirements for a function to be provided by the `generic object manager` are as follows:

1. The `generic object manager` only has to provide object type-independent functions that can be performed on all objects and must hence be shared by all `specific object managers` even though their implementation may differ.

2. The `generic object manager` must contain only those functions that are necessary for correctly enforcing a security policy. Thus, all other functions that are typically provided by object managers and that are not relevant for correctly enforcing security policies may not be part of the `generic object manager`. By this means, we reduce the probability that errors in functions irrelevant for policy execution have an impact on the enforcement of the security policies integrated in a causal TCB.

In the following, we motivate and present the functions provided by the `generic object manager`. Just like other functional components of the policy-independent RTE it provides two types of functions: management functions and functions to be performed on `objects`; we discuss management functions first.

As mentioned above, one significant prerequisite for establishing `object` authenticity is the unambiguous and irrevocable binding of the EIDs to the `objects`. Otherwise we cannot guarantee that the credentials, e.g., rights or attributes, which are assigned by a policy to the EIDs, are mapped to the correct `objects`. In this case, users could gain rights for other `objects` than intended by a policy, which then results in undermining the policy and thus violating the second requirement. For this reason, `createObject()` is provided by the `generic object manager`, which creates an `object` and binds its EID. Input is an EID generated by the `entity identification server`; the EID must first be requested by a security policy and is then given to the `generic object manager` via the `interceptor`. `Object_T` is the abstract type representing `objects`. Note that in a TCB's implementation, the `specific object manager` that implements persistent `objects` must bind the EIDs to the `objects` in a permanent way.

```
public T_Object createObject(EID_T entityID)
public void destroyObject(EID_T objectID)
```

Besides `object` creation, destroying `objects` is also relevant for maintaining the `objects'` authenticity and hence for the correct enforcement of the security policies. Supposing that the deletion of `objects` does not reside in the TCB. Errors in the 'delete' function may then lead to that either an `object` is not destroyed even though the policy considers it as destroyed, or an `object` is destroyed though a policy does not intend it to be destroyed. In the first case, the `entity identification server` will provide the EID of the destroyed `object` to a new `object` eventually, resulting in that this EID may be bound to more than one `object` by the `object manager`. This leads to that the new object gains the credentials of the destroyed `object` so that the security policy will be undermined. Hence, the second requirement is violated. The second case may influence the correct enforcement of the policy in the mediate term. Even though the `object` is destroyed, the `entity identification server` cannot give the `object's` EID to another `object` since it was not told that the `object` does not exist anymore. If this happens too often and depending on the range of the EID space, it may happen that the `entity identification server` is not able to produce new EIDs for new `objects`. Thus, the OS cannot run correctly anymore since the policy is not able to create new entities. On this account, the `generic object manager` provides the function `destroyObject()` for destroying an `object` and releasing its EID.

On top of these management functions, the `generic object manager` provides the abstract functions `read()` and `write()` to be performed on the `objects`. The reasons for including these functions in the policy-independent RTE are as follows. The responsibility of security policies is to protect the confidentiality and integrity of their entities. In case the policy allows an `object` to be read or written to, the policy requires that i) only the requesting `object` is able to read or write this `object` and ii) the addressed `object` is the `object` for which the requester is allowed to do so. For example, when a process acting on behalf of a user is allowed to write confidential information to a file, the `generic object manager` must ensure that this information is written to the correct file and not to another `object`, e.g., another file or even a network device. Reading `objects` follows the same argumentation. On this account, the `generic object manager` provides the function `read()`, enabling to read the data of an `object`, and the function `write()` that writes the data of an `object`.

```
public Data_T read(EID_T objectID)
public void write(EID_T entityID, Data_T data)
```

`Objects` usually have additional attributes, e.g., parent ID of a process, socket type, or file system type, depending on their object types. While the EIDs of `objects` as well as the data contained by `objects` are the only attributes significant for correctly enforcing policies, all other attributes are irrelevant for policy enforcement. Thus, they do not have to be managed within the policy-independent RTE so that attribute managing functions like getting or setting an `object's` attributes are to be implemented outside the TCB. In case one of these functions is erroneous, this may result in that object type-specific OS services are also erroneous or even not available. However, it does not influence the correct enforcement of the policies.

4.3.10 Authenticator

The functional component `authenticator` can be considered a specific object manager that is responsible for verifying the identity of the `objects` representing human users. It implements an authentication protocol to prove the identity of the OS's users by checking an authentication credential that can only be provided by the according user. Depending on the TCB implementation, authentication credentials can be passwords, certificates, or biometric attributes.

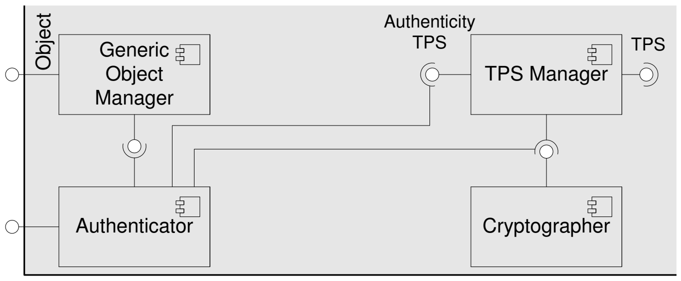

Figure 4.16: Authenticator

Figure 4.16 shows the `authenticator` and adjacent functional components with which it is related. The `authenticator` provides one interface, which represents the second entrypoint to the TCB. Before a user is allowed to perform any action in the OS, the user must verify its identity by presenting the requested credential. Here, the interface function `checkCredential()` must be called, requiring the EID of the user to be authenticated and the credential. Depending on the kind of credential, the `authenticator` must use cryptographic functions that are provided by the `cryptographer`.

```
public Boolean_T checkCredential(EID_T user, Credential_T cred)
```

The `authenticator` manages a policy-independent data structure *credential map* which maps every entity identifier to a user-specific credential by establishing a function $f : EID\_T \rightarrow Credential\_T$. Only if the presented credential equals the one stored in the *credential map* for the presented EID, the function returns `true`. The inputed user ID must equal an EID managed by the `entity identification server`.

When a new `object` is created by the `object manager` and this `object` represents a human user, the `object manager` calls the `authenticator` for inserting the user's credential in the *credential map*. For this purpose, the `authenticator` provides a function `addUser()`, requiring the user's EID and credential. Once a pair of EID and credential is inserted in the *credential map*, a user is enabled to authenticate herself. If a user is to be removed, her credential also has to be removed. Here, the `object manager` calls `deleteUser()` provided by the `authenticator`.

```
protected void addUser(EID_T user, Credential_T cred)
protected void deleteUser(EID_T user)
```

Just as the EIDs of the users, the users' credentials have to be stored permanently. The `AuthenticityTPS`, which already stores the EIDs, is also responsible for permanently keeping the *credential map*. When an authentication request is received by the `authenticator`, it retrieves the original user credential from the `AuthenticityTPS` and checks whether both credentials are equal. When a new user is created, the `authenticator` must save the new entry of the *credential map* in the `AuthenticityTPS`.

A common feature of software systems that implement a credential-based authentication protocol is to enable the users to change their authentication credentials, e.g., in case they have forgotten their passwords or leaked their certificates. Causal TCBs do not provide a specific function for changing the credential due to the following reason. Changing the credential is a convenient function, but it is not directly essential for correctly enforcing a security policy. In case an authentication credential really needs to be changed for some security reason though, an administrator can first delete the user and then add a new user with the same EID. By this means, the administrator is able to provide a new authentication credential for the same EID.

4.3.11 Summary

This section has motivated and presented the functional components of the policy-independent RTE along with a detailed discussion of their functions. All functions are characterized by well-structured signatures and well-defined responsibilities. The policy-independent RTE contains ten functional components that provide eight abstractions, all of which are required by any real world-based causal TCB for protecting and enforcing its security policies. The functional perimeter of each component is straightforward and clearly laid out; functional redundancies are avoided by applying microkernel-based approaches. While each functional component provides a set of interface functions to be used only by security policies or components of the policy-independent RTE, the RTE provides two dedicated interfaces that serve as entrypoints to the TCB and can be used by an OS. These public interfaces are well-defined with low complexity, providing a total of five entrypoint functions.

The following sections introduce the policy-dependent RTE along with a method to derive its functions. The latter is based on a formal rule system that allows for deriving necessary functions of the policy-dependent RTE from a TCB's security policies.

4.4 Policy-dependent Runtime Environment

As motivated in Section 4.2, the policy-dependent RTE consists of two sublayers, one of which contains security model functions, while the second one contains model instance functions. Figure 4.17 gives an overview of the functional components of the policy-dependent RTE.

The `interceptor` represents the set of model instance functions (top layer of the functional TCB design), because it is policy-specific. As implied by its name, the functional component `security model functions` implements the security models by which a TCB's policies are formalized; it represents the middle layer of the functional TCB design. `Security model functions` are directly derived from the policies by an engineering method that exploits causal dependencies between policies and TCB functions.

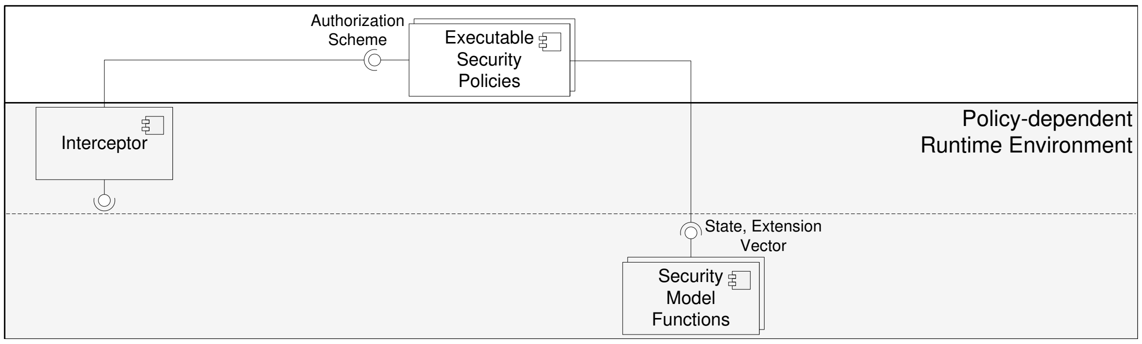

Figure 4.17: Functional Components of the Policy-dependent RTE

The goal of this section is to present the functions of the policy-dependent RTE. For this purpose, we introduce an engineering method for the policy-dependent RTE and show how TCB functions are derived from a security policy. In Section 4.4.1, we present the idea and the approach of engineering the policy-dependent RTE and discuss important design decisions. Sections 4.4.2 and 4.4.3 then present the engineering method in an abstract way that is independent of any model and policy; here, we discuss the functions of the functional components `interceptor` and `security model functions` by drawing an informal link between core-based models and TCB functions. Afterwards, Section 4.4.4 formalizes this link by specifying a formal rule system that defines necessary and sufficient conditions for policy-dependent TCB functions.

4.4.1 Engineering Approach

The idea of engineering a TCB's policy-dependent RTE is to derive the required TCB functions from a TCB's security policies by exploiting causal dependencies between policies and TCB functions. Thereby, we aim at specifying a formal rule system, in order to provide the basis for an automated TCB composition tool. To this end, a sound theoretical foundation is required, which we establish by applying a formal algebraic approach. Core-based security models are then formally described by a set of algebras and hence on a more implementation-oriented level of abstraction. This results in a much smaller semantical gap between formal models and implementation-oriented TCB functions such that bridging this gap is less error-prone. Note that a policy engineer does not need to know the algebraic fundamentals of TCB engineering. These are exclusively required to specify a formal rule system and to develop a tool that automatically composes the policy-dependent RTE.

Following an algebraic approach, TCB engineering consists of two steps as shown in Figure 4.18. The first step is to rewrite a core-based model in algebraic notation (*algebra engineering*). The rewriting does not impose a semantical change; rather, it is an object-oriented reorganization of a model's components where for instance primitive actions are directly attached to the model's state. This provides the basis for deriving TCB functions by a rule

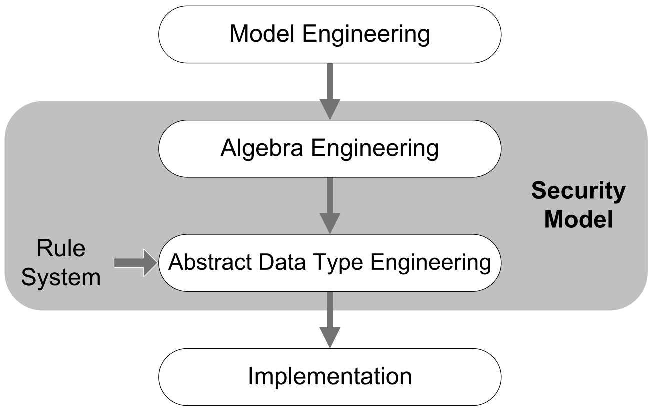

Figure 4.18: Engineering the Policy-dependent RTE

system in the second step; result is a set of TCB functions, which are denoted by *abstract data types (ADTs)* to describe the functions' syntax and semantics (*abstract data type engineering*). This approach matches the idea of deriving implementation-independent TCB functions that can be implemented on varying platforms, instead of aiming at a specific platform-dependent algebra implementation. Core-based models provide all required operations of an algebra and thus define the interpretation of an ADT's operators.

In general, there are two approaches for determining the policy-dependent functions of a causal TCB. Functions can be derived from

(i) either a policy itself, i.e., from the policy's model instance, or

(ii) the model.

The advantage of the first approach is that a TCB enforcing a single policy can be precisely tailored to the functional requirements of this policy; its functional range then is minimal with respect to this policy. On the other hand, in case a TCB enforces multiple policies at the same time, functional redundancies may arise. The argumentation for the second approach is vice versa; while the functional range of a single-policy TCB may not be minimal with respect to this policy, functional redundancies in multi-policy TCBs can be avoided. For example, let us consider two arbitrary IBAC model instances that share the same state definition $q = (S_q, O_q, m_q)$, consisting of a subject set, an object set, and an ACM, but differ in their sets of primitive actions. While the second instance can add and delete subjects and objects as well as add rights to and delete rights from the ACM (six primitive actions), let us assume that the authorization scheme of instance one does not add new subjects or objects but applies only the other four primitive actions. For the sake of straightforwardness, all other model components are equivalently defined.

Following the first approach results in two different `security model functions` components, one of which provides the functions for one instance. The first component then contains among others TCB functions that implement the policy state and the four primitive actions. The second `security model functions` component supports the second instance by containing functions to implement the state and the six primitive actions, including the four primitives which are also shared by both instances. Already this small example shows that functional redundancies may arise, which contravenes with the TCB design requirements discussed in Section 4.1.2.

When deriving TCB functions from models instead of model instances, it still may happen that functional redundancies arise. However, when following this approach we are able to avoid them a priori by applying so called *subsuming security models*. A subsuming security model can be considered an upper bound for the functional perimeter of a policy-dependent RTE. A subsuming model has the same state and extension vector definition as the subsumed models and combines all primitive actions and clauses of these models. Referring to the above example, the model underlying the second policy instance can be considered a subsuming model because is contains all six primitive actions. TCB functions are then derived exclusively from the second model, resulting in a single `security model functions` component that supports both models. Following this approach, we avoid functional redundancies in multi-policy TCBs; each `security model functions` component contains functions to implement one specific subsuming model, which then combines various model instances. On the other hand, the functional range of a single-policy TCB may not be minimal with respect to this policy. However, since avoiding functional redundancies is more important for this dissertation, we decide on following the second approach of deriving TCB functions based on subsuming models.

It remains to discuss how subsuming models are determined. When engineering a TCB with multiple policies, we combine all models with the same state and extension vector definitions. In doing so, their sets of primitive action sets as well as their condition sets are united while all other model components remain. Result is a set of subsuming models; in the best case the set contains only one model, in the worst case the number of subsuming models equals the number of different model instances. When a new policy is added to a TCB, we have to check whether there already is a subsuming model for this policy. If so, the subsuming model and thus its `security model functions` component may be enhanced by new primitive actions and conditions. Otherwise, a new subsuming model based on the given model instance has to be defined, resulting in an additional `security model functions` component.

The following section shows how to derive the functions of the `security model functions` component.

4.4.2 Abstract Security Model Functionality

The goal of this section is to present the engineering method for a TCB's policy-dependent RTE. Before discussing it in detail, we provide a formal definition of the term algebra in connection with abstract data types based on [75]. Section 4.4.2.1 then shows how core-based models are described by algebras. Afterwards, we present the functions of the functional component `security model components` that are derived from said algebras in Section 4.4.2.2. Both is done in an abstract way that is independent of any specific model.

An algebra is a pair $A = (\mathcal{S}_A, \mathcal{O}_A)$ with a set $\mathcal{S}_A$ of base sets and a set $\mathcal{O}_A$ of operations. The domain of each operation is defined as the (potentially empty) Cartesian product of a subset of $\mathcal{S}_A$. The codomain of each operation is defined by exactly one base set. For each base set there exists at least one operation where the base set either belongs to the domain or defines the codomain of the operation.

Definition 4.4 (Algebra) An *algebra* is a tuple $A = (\mathcal{S}_A, \mathcal{O}_A)$ where

- $\mathcal{S}_A$ is a finite set of base sets, and
- $\mathcal{O}_A = \{o_A \mid o_A : s_{1,A} \times \cdots \times s_{n,A} \to s_A\}$ is a finite set of operations.

Every ADT is based on a signature that contains the sorts of the involved data and operators that are defined on these sorts. For each sort there must be at least one operator where the sort is either part of the domain or specifies the codomain. While this signature defines the number and syntax of an ADT's operators, it does not define their interpretation. This is usually defined by an additional set of axioms.

Definition 4.5 (Abstract Data Type) An *abstract data type (ADT)* consists of a signature Σ and a set of axioms X. The signature $\Sigma = (\mathcal{S}, \mathcal{O})$ of an ADT is a tuple where

- $\mathcal{S}$ is a finite set of sorts, and
- $\mathcal{O} = \{o \mid o : s_1 \times \cdots \times s_n \to s\}$ is a finite set of operators.

The set of axioms describe the interpretation of every $o \in \mathcal{O}$.

ADT engineering now determines for each algebra operation, which now represents the functional specification of a TCB, those ADT operators that are required to implement the operation. This is possible due to the causal dependencies between security policies (represented by a set of algebras) and TCB functions (represented by ADT operators). The approach is to define a mapping $\varphi : \mathcal{O}_A \to \mathscr{P}(O)$, which maps each algebra operation to a set of ADT operators that are required for its implementation (called *basic ADT operator*). This mapping is then controlled by a rule system that formalizes causal dependencies between policies and TCB functions (Section 4.4.4). Besides, each algebra operation is mapped to an *interface ADT operator*. The resulting set of interface operators defines the interface functions of the `security model components`, which are implemented by the basic ADT operators.

4.4.2.1 Algebra Engineering

To rewrite a security model in algebraic notation, we follow two steps. In doing so, engineering algebras is similar to engineering core-based models, where we first define all model components, before specifying a models' state and extension vector (Section 3.3). Algebra engineering first defines algebras for all model components, before defining algebras for a model's state and extension vector. More precisely, algebra engineering defines

1. *Component Algebras* for all model components in M (Section 3.3).
2. *Core Algebras*, called *State* and *ExtensionVector*, that are based on said component algebras by attaching all primitive actions and clauses of a model to either the *State* or the *ExtensionVector* algebra.

The result is a set of algebras that contains a set of model-dependent component algebras and two core algebras. Core algebras represent the model abstractions state q and extension vector E and are thus called *State* and *ExtensionVector*. We do not provide an explicit representation for the state transition function; in fact a model's primitive actions are attached to *State*, clauses are attached to both core algebras. Component algebras represent the components of a model defined by M; they define the basic sorts of the algebras *State* and *ExtensionVector*, and the basic operations that can be performed on them. In the following, we show how component algebras and core algebras are defined.

Component Algebras In order to develop an efficient TCB engineering method with low complexity, component algebras have to meet three basic requirements:

- Reusability: Component algebras have to be universal to allow for describing a wide variety of model components of core-based models. This provides the basis for reusing them to engineer causal TCBs for a wide spectrum of policies.
- Conciseness: Component algebras must be concise and pragmatic. This contributes to reducing the degree of error-proneness of the overall TCB engineering method.
- Expandability: In order to provide support for future security policy and model trends, the component algebras have to be easily adaptable and expandable.

These requirements have motivated two fundamental design decisions. Instead of modeling an algebra for each model component, e.g., subject set, right set, or object set, we combine model components of the same mathematical type to *parameterized algebras*. That means for example, we design a component algebra *Set*[*Element*][3] that is parameterized by an element type, e.g., subjects, rights, and objects, instead of different algebras, e.g., *Subjectset*, *Rightset*, or *Objectset*. On this account, we meet the requirement of universality and thus reusability.

Based on parameterized algebras, all standard mathematical types that are typically used in core-based models can be modeled by a combination of two algebras that represent the mathematical types sets and tuples. For example, we can model a binary relation as an algebra *Set*[(*Domain*, *Target*)], which is a set of tuples where the first tuple element is the domain of the relation and the second element represents the codomain. This approach is the most universal to define component algebras. At the same time, however, it is intricate and error-prone when modeling more complex mathematical types like ACMs, which are then considered as left-total and right-unique 3-ary relations. The latter contravenes the second requirement so that we compromise between reusability and conciseness by specifying *an individual parameterized algebra for each mathematical type*. These algebras can still be reused for modeling different components, e.g., an algebra *Relation*[*Domain*, *Target*] can model a dominance relation as well as a role exclusion relation, while they are more pragmatic to work with. On top of that, new algebras can be designed, in case the algebra foundation is not sufficient; hence, the third requirement is also met.

In the following, we present four basic component algebras: *Set*[*Element*], *Relation*[*Domain, Target*], *Mapping*[*Domain, Target*], and *Matrix*[*Row, Column, Target*], representing the mathematical types set, binary relation, unary mapping, and a special binary mapping. These types have in common that they are very general and are used in numerous core-based models. They build an algebra hierarchy in that they, just like the corresponding mathematical types, are based on each other: the algebra *Set*[*Element*] builds the foundation for the algebras *Relation*[*Domain, Target*], *Mapping*[*Domain, Target*], and *Matrix*[*Row, Column, Target*]. The above mentioned algebras build a universal algebra foundation for TCB engineering that is sufficient to model a wide variety of models. Based on these algebras, Section 4.4.4 then provides a formal rule system and a set of predefined TCB functions. In case additional mathematical types are applied in models, e.g., 3-ary relations or lattices, additional component algebras can be specified, which then require to extend the formal rule system as well as the predefined function set.

[3]To ease the reading, all algebras and their operations will be written in *italic style* throughout this dissertation.

Any core-based model uses the mathematical type set to model elements of either the model state or the extension vector. Hence, the component algebra *Set*[*Element*] (short *Set*, Table 4.1) can be used for various model components described in algebraic notation.

import	$Bool \cong \{true, false\}$
	$Nat \cong \mathbb{N}$
	$Element \cong e :=$ element of a nonempty set of a security model, $e \in E$, where $E = \bigcup_i^n d_i \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} is a component of the model state $q = (d_{1_q}, \ldots, d_{n_q})$, e_j is an element of the extension vector $E = (e_1, \ldots, e_h)$, and d_{i_q} and e_j are of mathematical type set
sorts	$Set \cong S \subseteq E, S$ is finite
operations	$s\_\varnothing :\rightarrow Set$
	$s\_insert : Set \times Element \rightarrow Set$
	$s\_\in : Set \times Element \rightarrow Bool$
	$s\_\cup : Set \times Set \rightarrow Set$
	$s\_\backslash : Set \times Set \rightarrow Set$
	$s\_numElements : Set \rightarrow Nat$
semantics	$s\_\varnothing() := \varnothing$
	$s\_insert(S, s) := S \cup \{s\}$
	$is\_\in(S, s) := \begin{cases} true, & s \in S \\ false, & s \notin S \end{cases}$
	$s\_\cup(S_1, S_2) := S_1 \cup S_2$
	$s\_\backslash(S_1, S_2) := S_1 \backslash S_2$
	$s\_numElements(S) := \lvert S \rvert$

Table 4.1: Abstract Component Algebra $Set[Element] = (\mathcal{S}_{A_S}, \mathcal{O}_{A_S})$

Set[*Element*] imports the sorts *Bool* and *Nat*[4] and defines two sorts *Element* and *Set*. While the sorts are independent of any model, their base sets and operations depend on an underlying model. In general it holds that the base set of the sort *Element* may be defined by any component of a model's state or extension vector that is of mathematical type set. *Set* then is defined as a finite subset of this base set. The set of operations is motivated by a model's primitive actions and clauses. More precisely, if an algebra describes a dynamic model component that is part of the model's state, its operation set contains projector operations (including constructors) as well as selector operations. In case it describes a static model component, the operation set only contains selector and constructor operations. For example, the HRU primitive *createSubject* adds a new subject to the subject set, modeled by the set operation $\cup$. Consequently, to describe the subject set in algebraic notation, an operation with the same interpretation (called $s\_\cup()$) must be contained by the operation set. Table 4.1 illustrates an abstract component algebra for sets that is independent of any specific model. Besides the constructors $s\_\varnothing()$ and $s\_insert()$, it contains standard set operations like $\in$, $\cup$, $\backslash$, or *cardinal number*, which are typically used by models that have sets as state components.

To complete the specification of any parameterized algebra, we also have to provide specifications for each applied parameter sort, here *Element* as illustrated by Table 4.2. This specification defines the sort *Element* and one operation, that enables to construct any ob-

[4]Due to completeness reasons, we need to define algebras for the sort *Bool* and *Nat*. Since such specifications can be found in the literature numerous times, e.g., in [75,76], we refrain from doing so. In this dissertation we refer to the base sets $Bool = \{true, false\}$ and $Nat = \mathbb{N}$ with standard operations *true*, *false*, $\neg$, $\wedge$, and $\vee$ for *Bool*, and 0, *succ*, and *add* for *Nat*.

ject of sort *Element*. This is a very basic specification for a parameter sort; security models typically require additional operations, e.g., to compare two elements for equality.

sorts	$Element \cong e :=$ element of a nonempty set (Universe)
operations	$create :\to Element$
semantics	$create() = e$

Table 4.2: Abstract Parameter Sort *Element*

As already mentioned, the algebra *Set* provides the basis for the component algebra *Relation*[*Domain*, *Target*] that models binary relations (short *Relation*, Table 4.3). It imports the sorts *Bool*, *Set*[*Element*], *Domain*, and *Target*, where *Domain* and *Target* are parameter sorts like *Element*, and defines the sort *Relation*. Analogous to the algebra *Set*, the sorts are model-independent; only the base sets of the parameter sorts *Domain* and *Target* as well as the operation set depend on a specific model. More precisely, the base sets for *Domain* and *Target* are defined by the model's state and extension vector components; the operations are based on the model's primitive actions and clauses. The base set of *Domain* is a unification of all domains of relation components of a model; analogously the base set of *Target* unifies all codomains of model components that are of type relation. Note that we also have to define algebra specifications for *Domain* and *Target*; however, since these are equal to the specification of *Element*, we refrain from doing so. Table 4.3 illustrates the abstract component algebra *Relation*[*Domain*, *Target*], where we have defined its operations based on standard operations that are frequently used in core-based models. As can be seen, the operation set of *Relation* is a subset of the operation set of *Set* due to their mathematical similarities. On top of that, it contains the frequently used operations $\lhd, \rhd, \ntriangleleft, \ntriangleright$, and $\ntriangleright$.

import	$Bool \cong \{true, false\}$ $Domain \cong d :=$ element of a nonempty set of a security model, where $D = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} is a component of the model state $q = (d_{1_q}, \ldots, d_{n_q})$, e_j is an element of the extension vector $E = (e_1, \ldots, e_h)$, and d_{i_q} and e_j are of mathematical type set $Target \cong t :=$ element of a nonempty set of a security model, where $T = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} and e_j as defined above $Set[Element] \cong S \subseteq D \cup T, S$ is finite
sorts	$Relation \cong R \subseteq D \times T, Rel$ is finite
operations	$r\_\varnothing :\to Relation$ $r\_insert : Relation \times Domain \times Target \to Relation$ $r\_\in: Relation \times Domain \times Target \to Bool$ $r\_\cup : Relation \times Relation \to Relation$ $r\_\setminus : Relation \times Relation \to Relation$ $r\_\lhd : Relation \times Set[Domain] \to Relation$ $r\_\rhd : Relation \times Set[Target] \to Relation$ $r\_\ntriangleleft : Relation \times Set[Domain] \to Relation$ $r\_\ntriangleright : Relation \times Set[Target] \to Relation$
semantics	$r\_\varnothing() := \varnothing$ $r\_insert(R, d, t) := R \cup \{(d, t)\}$ $r\_\in (R, d, t) := \begin{cases} true, & (d,t) \in R \\ false, & (d,t) \notin R \end{cases}$ $r\_\cup (R_1, R_2) := R_1 \cup R_2$

$r\_\backslash(R_1, R_2) := R_1 \backslash R_2$
$r\_\lhd(R, \{d_1, \dots, d_i\}) := \{(d, t) \mid (d, t) \in R \wedge d \in \{d_1, \dots, d_i\}\}$
$r\_\rhd(R, \{t_1, \dots, t_j\}) := \{(d, t) \mid (d, t) \in R \wedge t \in \{t_1, \dots, t_j\}\}$
$r\_\ntriangleleft(R, \{d_1, \dots, d_i\}) := \{(d, t) \mid (d, t) \in R \wedge d \notin \{d_1, \dots, d_i\}\}$
$r\_\ntriangleright(R, \{t_1, \dots, t_j\}) := \{(d, t) \mid (d, t) \in R \wedge t \notin \{t_1, \dots, t_j\}\}$

Table 4.3: Abstract Component Algebra $\mathit{Relation}[\mathit{Domain}, \mathit{Target}] = (\mathcal{S}_{A_R}, \mathcal{O}_{A_R})$

Besides sets and relations, unary and binary mappings are also frequently used in core-based models. Since mappings are relations with special properties (left-total and right-unique), we may reuse the algebra *Relation* and add operations that are specific to mappings. However, this contravenes the second requirement by overloading one algebra with two meanings. For this reason, we decide on designing an extra algebra called *Mapping*[*Domain, Target*] (short *Mapping*), which represents a left-total and right-unique binary relation (Table 4.4). Consequently, it is not surprising that the algebras *Relation* and *Mapping* have many common characteristics regarding their sorts, base sets, and operations. As *Relation*, the base sets of the sorts *Domain* and *Target* are defined by a model's state and extension vector components, while all other sorts are model-independent. Note that the base set of *Target* may also be defined by the power set of some state or extension vector component; this depends on the codomains of the model components (an example is given in Section 4.5.1). Compared to the specification of *Relation*, the main difference is the definition of the modifying operations ($r\_\cup()$ and $mp\_\cup()$, $r\_insert()$ and $mp\_insert()$) since these are responsible for establishing the properties left-total and right-unique. All other operations are equally defined. Additionally, mappings may also have specific operations, e.g., functional overriding and target selection of a given domain element, as defined by $mp\_\oplus()$ and $mp\_getMapping()$.

import $\mathit{Bool} \cong \{\mathit{true}, \mathit{false}\}$
$\mathit{Domain} \cong d :=$ element of a nonempty set of a security model, where
$D = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} is a component of the model state $q = (d_{1_q}, \dots, d_{n_q})$, e_j is an element of the extension vector $E = (e_1, \dots, e_h)$, and d_{i_q} and e_j are of mathematical type set
$\mathit{Target} \cong t :=$ element of a nonempty set of a security model, where
$T = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j \cup \bigcup_i^n \mathscr{P}(d_{i_q}) \cup \bigcup_j^h \mathscr{P}(e_j), 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} and e_j as defined above
$\mathit{Set}[\mathit{Element}] \cong S \subseteq D \cup T, S$ is finite

sorts $\mathit{Mapping} \cong \mathit{Map} \subseteq D \times T, \mathit{Map}$ is finite

operations $mp\_\varnothing() : \mathit{Mapping}$
$mp\_insert : \mathit{Mapping} \times \mathit{Domain} \times \mathit{Target} \to \mathit{Mapping}$
$mp\_\in: \mathit{Mapping} \times \mathit{Domain} \times \mathit{Target} \to \mathit{Bool}$
$mp\_getMapping : \mathit{Mapping} \times \mathit{Domain} \to \mathit{Set}[\mathit{Target}]$
$mp\_\cup : \mathit{Mapping} \times \mathit{Mapping} \to \mathit{Mapping}$
$mp\_\backslash : \mathit{Mapping} \times \mathit{Mapping} \to \mathit{Mapping}$
$mp\_\lhd : \mathit{Mapping} \times \mathit{Set}[\mathit{Domain}] \to \mathit{Mapping}$
$mp\_\rhd : \mathit{Mapping} \times \mathit{Set}[\mathit{Target}] \to \mathit{Mapping}$
$mp\_\ntriangleleft : \mathit{Mapping} \times \mathit{Set}[\mathit{Domain}] \to \mathit{Mapping}$
$mp\_\ntriangleright : \mathit{Mapping} \times \mathit{Set}[\mathit{Target}] \to \mathit{Mapping}$
$mp\_\oplus : \mathit{Mapping} \times \mathit{Mapping} \to \mathit{Mapping}$

semantics $mp\_\varnothing() := \varnothing$

$$mp\_insert(\mathit{Map}, d, t) := \begin{cases} \mathit{Map} \cup \{(d, t)\}, & \nexists (d', t') \in \mathit{Map} : d = d' \\ \mathit{Map}, & \mathit{otherwise} \end{cases}$$

$$
\begin{aligned}
&mp\_\in(Map, d, t) := \begin{cases} true, & (d,t) \in Map \\ false, & (d,t) \notin Map \end{cases} \\
&mp\_getMapping(Map, d) := \begin{cases} \{t\}, & (d,t) \in Map \\ s\_\varnothing(), & (d,t) \notin Map \end{cases} \\
&mp\_\cup(Map, Map') := \begin{cases} Map \cup Map', & \nexists (d,t) \in Map, (d',t') \in Map' : d = d' \\ & \wedge t \neq t' \\ Map, & otherwise \end{cases} \\
&mp\_\backslash(Map, Map') := Map \backslash Map' \\
&mp\_\lhd(Map, \{d_1, \ldots, d_i\}) := \{(d,t) \mid (d,t) \in Map \wedge d \in \{d_1, \ldots, d_i\}\} \\
&mp\_\rhd(Map, \{t_1, \ldots, t_j\}) := \{(d,t) \mid (d,t) \in Map \wedge t \in \{t_1, \ldots, t_j\}\} \\
&mp\_\ntriangleleft(Map, \{d_1, \ldots, d_i\}) := \{(d,t) \mid (d,t) \in Map \wedge d \notin \{d_1, \ldots, d_i\}\} \\
&mp\_\ntriangleright(Map, \{t_1, \ldots, t_j\}) := \{(d,t) \mid (d,t) \in Map \wedge t \notin \{t_1, \ldots, t_j\}\} \\
&mp\_\oplus(Map, Map') := \{(d,t) \mid (d,t) \in Map \wedge d \notin D'\} \cup Map', \text{ where } D' \text{ is the} \\
&\qquad\qquad\qquad\qquad \text{domain of } Map'
\end{aligned}
$$

Table 4.4: Abstract Component Algebra $Mapping[Domain, Target] = (\mathcal{S}_{A_{Map}}, \mathcal{O}_{A_{Map}})$

A special case of mappings that is frequently used in core-based models are ACMs. As demonstrated by means of examples in Section 2.3.2, ACMs are generally left-total and right-unique binary mappings $acm : X \times Y \rightarrow 2^Z$, which map a pair (x, y) to a set $z \in 2^Z$. In this context, X represents the matrix' rows, Y represents the columns, and 2^Z is the type of the cells' content. When designing an algebraic representation for ACMs, we have two alternatives. (i) We reuse the algebra *Mapping*[*Domain, Target*], redefine the sort *Domain* as a tuple where its elements are of row and of column type, and specify additional matrix-specific operations. The disadvantage of this approach is that we cannot specify operations explicitly defined on cells or rows, e.g., for their creation or deletion, since there are no individual sorts for rows and columns. However, such operations are significant because they are frequently used in core-based models. On top of that, such an algebra is more intricate and does not contribute to meet the conciseness requirement. (ii) We specify an additional component algebra *Matrix*[*Row, Column, Target*] as explicit algebraic representation of an ACM. Here, columns, rows, and cells are defined by individual sorts, which allow for specifying column- and row-specific operations. However, this results in additional efforts. Nevertheless, we follow the second approach for two reasons. First, ACMs are used in almost any core-based model such that the high degree of reusability of this component algebra justifies the additional effort. Second, besides row- and column-specific operations, an individual algebra is also more pragmatic to use.

Table 4.5 illustrates the abstract component algebra *Matrix*[*Row, Column, Target*] (short *Matrix*). As can be seen, it imports the sorts *Bool*, *Row*, *Column*, *Target*, and *Set*[*Element*], where *Row*, *Column*, and *Target* are parameter sorts. The sort *Matrix* is independent of a model; it is defined as a 3-ary relation, where the cell content is typed by the power set of a set of target elements[5]. This enables a straight forward definition of the *m_insert*() operation, which is responsible for ensuring the properties left-total and right-unique. In case a cell does not yet exist, the cell is added or an element is inserted into an already existing cell. The base sets of the sorts *Row*, *Column*, and *Target* are defined by a model's state and extension vector components. While it is common for core-based models to use dynamic and static components to define the types of their ACMs' rows and columns, e.g., subject and object sets of HRU models, the cells' content is usually typed by static model components

[5] A special case of this specification are target sets with one element, representing $acm : X \times Y \rightarrow Z$. By this means, the specification is general and open to an even wider spectrum of security models.

such as a right set. However, the algebra *Matrix* allows for a dynamic cell type since this is more general and allows to support a greater variety of models and policies. While the operation set of the component algebra is model-specific, too, the abstract algebra's set of operations is inspired by typical operations on matrices used by core-based security models. For example, $m\_\in()$ checks whether a target element such as a right is in a specific cell, and $m\_cellContent()$ selects all target elements of a specific cell. In case rows and columns are defined by state rather than extension vector components, operations such as $m\_addRow()$, $m\_deleteRow()$ or $m\_addColumn()$, $m\_deleteColumn()$ can be applied.

import	$Bool \cong \{true, false\}$
	$Row \cong r :=$ element of a nonempty set of a security model, where
	$R = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} is a component of the model state $q = (d_{1_q}, \ldots, d_{n_q})$, e_j is an element of the extension vector $E = (e_1, \ldots, e_h)$, and d_{i_q} and e_j are of mathematical type set
	$Column \cong c :=$ element of a nonempty set of a security model, where
	$C = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} and e_j as defined above
	$Target \cong t :=$ element of a nonempty set of a security model, where
	$T = \bigcup_i^n d_{i_q} \cup \bigcup_j^h e_j, 1 \leqslant i \leqslant n, 1 \leqslant j \leqslant h$, with d_{i_q} and e_j as defined above,
	$Set[Element] \cong S_T \subseteq T$, S is finite
sorts	$Matrix \cong M \subseteq R \times C \times 2^T, M$ is finite
operations	$m\_\varnothing : \rightarrow Matrix$
	$m\_init : Matrix \rightarrow Matrix$
	$m\_insert : Matrix \times Row \times Column \times Target \rightarrow Matrix$
	$m\_\in : Matrix \times Row \times Column \times Target \rightarrow Bool$
	$m\_cellContent : Matrix \times Row \times Column \rightarrow Set[Target]$
	$m\_delete : Matrix \times Row \times Column \times Target \rightarrow Matrix$
	$m\_addRow : Matrix \times Row \rightarrow Matrix$
	$m\_deleteRow : Matrix \times Row \times Target \rightarrow Matrix$
	$m\_addColumn : Matrix \times Column \rightarrow Matrix$
	$m\_deleteColumn : Matrix \times Column \rightarrow Matrix$
semantics	$m\_\varnothing() := \varnothing$
	$m\_init(M) := \{(r, c, t) \mid (r, c, t) \in M\}$
	$m\_insert(M, r, c, t) := \begin{cases} (r', c', S_T \cup \{t\}), & (r', c', S_T) \in M : r = r' \wedge c = c' \\ M \cup \{(r, c, \{t\})\}, & \nexists (r', c', t') \in M : r = r' \wedge c = c' \end{cases}$
	$m\_\in(M, r, c, t) := \begin{cases} true, & (r', c', S_T) \in M \wedge r = r' \wedge c = c' \wedge t \in S_T \\ false, & otherwise \end{cases}$
	$m\_cellContent(M, r, c) := \begin{cases} S_T, & \exists! (r', c', S_T) \in M : r = r' \wedge c = c' \\ \varnothing, & otherwise \end{cases}$
	$m\_delete(M, r, c, t) := \begin{cases} M \backslash \{(r', c', S_T)\} & \\ \cup \{(r', c', S_T \backslash \{t\})\}, & (r', c', S_T) \in M : r = r' \wedge c = c' \\ M & \end{cases}$
	$m\_addRow(M, r') := M \cup \{(r', c_i, \varnothing) \mid 1 \leqslant i \leqslant \lvert C \rvert\}$
	$m\_deleteRow(M, r') := M \backslash \{(r, c, S_T) \mid r = r'\}$
	$m\_addColumn(M, c') := M \cup \{(r_j, c', \varnothing) \mid 1 \leqslant j \leqslant \lvert R \rvert\}$
	$m\_deleteColumn(M, c') := M \backslash \{(r, c, S_T) \mid c = c'\}$

Table 4.5: Abstract Component Algebra $Matrix[Row, Column, Target] = (\mathcal{S}_{A_M}, \mathcal{O}_{A_M})$

Using these component algebras *Set*, *Relation*, *Mapping*, and *Matrix*, we are able to describe a wide spectrum of model components in algebraic notation. Moreover, we can easily define additional component algebras, e.g., for lattices, if the algebra foundation is not sufficient. Here, the same design principles can be applied, which leads to that the novel algebra is nested in the algebra hierarchy so that sorts and operations of the existing algebras can be reused. We refrain from defining additional abstract component algebras here, since it is not our intention to cover every possible special case but develop a universal method for engineering a TCB's policy-dependent RTE.

Core Algebras Based on the component algebras, it remains to specify the core-specific algebras *State* and *ExtensionVector*. Again, we will provide a general method along with abstract algebras that have to be specialized for a specific model.

As mentioned earlier, the algebra *State* represents the model state $q \in Q$. Now, all component algebras that represent the state components of a model must be imported as sorts, in order to specify the sort *State* as an n-tuple of the imported sorts. The algebra's operation set is composed by a model's set of primitive actions *PRIM* and those clauses, which are defined on state components. If there is at least one such clause, the sort *Bool* must be imported. The type of state-defined clauses is generally defined as $clause : Q \times X \rightarrow Bool$, where the state set acts as an input parameter type. All remaining clauses are of type $clause : X \rightarrow Bool$. This means, they are defined on static components and hence belong to the operation set of the *ExtensionVector* algebra.

Table 4.6 illustrates the abstract algebra *State*; note that the imported sorts as well as the input parameters of the operations are examples since their number and type are model-specific. In the following, we provide more details on how to define the operations and their semantics.

import	$Bool \cong \{true, false\}$
	$Element \cong e :=$ element of a nonempty set of a security model
	$Domain \cong d :=$ element of a nonempty set of a security model
	$Target \cong t :=$ element of a nonempty set of a security model
	$\dots$
	$Set[Element] \cong S := \{s_1, \dots, s_n \mid n \in \mathbb{N}\}$
	$Relation[Domain, Target] \cong R \subseteq \{D \times T \mid R \, is \, finite\}$
	$Mapping[Domain, Target] \cong Map \subset \{D_1 \times T_2 \mid Map \, is \, finite\}$
	$Matrix[Row, Column, Target] \cong M \subset \{R \times C \times T_3 \mid M \, is \, finite\}$
	$\dots$
sorts	$State \cong q := (S, R, Map, M, \dots)$
operations	$createState : Set[Element] \times Relation[Domain, Target] \times Mapping[Domain, Targte] \times Matrix[Row, Column, Target] \rightarrow State$
	$getSetElement : State \rightarrow Set[Element]$
	$getRelationElement : State \rightarrow Relation[Domain, Target]$
	$getMappingElement : State \rightarrow Mapping[Domain, Target]$
	$getMatrixElement : State \rightarrow Matrix[Row, Column, Target]$
	$\dots$
	$prim_1 : State \times Set[Element] \rightarrow State$
	$prim_2 : State \times Set[Element] \rightarrow State$
	$\dots$
	$prim_l : State \times Element \times Element \rightarrow State$
	$clause_1 : State \times Relation[Domain, Target] \rightarrow Bool$
	$clause_2 : State \times Mapping[Domain, Target] \rightarrow Bool$
	$\dots$
	$clause_m : State \times Domain \times Target \rightarrow Bool$

semantics	$\mathit{createState}(S, R, \mathit{Map}, M) := (S, R, \mathit{Map}, M)$
	$\mathit{getSetElement}(q) := S$
	$\mathit{getRelationElement}(q) := R$
	$\mathit{getMappingElement}(q) := \mathit{Map}$
	$\mathit{getMatrixElement}(q) := M$
	...
	$\mathit{prim}_1(q, S)$
	$\mathit{prim}_2(q, S)$
	...
	$\mathit{prim}_l(q, s_i, s_j)$
	$\mathit{clause}_1(q, R)$
	$\mathit{clause}_2(q, \mathit{Map})$
	...
	$\mathit{clause}_m(q, d, t)$

Table 4.6: Abstract Core Algebra *State*

The operation set consists of four operation types: constructors, selectors, and operations that represent primitive actions and clauses. Out of these, only the constructors and selectors are not part of a model; they are exclusively required for a model's implementation. The constructor *createState*() is necessary to construct an object of the sort *State* and to initialize all of its tuple elements. For this reason, the constructor requires representatives of the tuple elements as input parameters. Additionally, the selector operations, e.g., *getSetElement*(), are required to select specific tuple elements of the *State*. Due to type safety, we provide an operation for each element of the extension vector tuple. In the following, we discuss how to derive the operations of *State* that represent a model's primitive actions and clauses.

As described in Section 3.2, the model core defines the input parameters of primitive actions and clauses by the input parameter vector $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$, where $T = \{D_1, ..., D_n, d_{1_q}, ..., d_{n_q}, e_1, ..., e_h, 2^{e_1}, ..., 2^{e_h}\}$. The algebraic representation differs from this notation in that it does not define an input parameter vector that is typed by all possible components of a model's state set, state, and extension vector, but individual input parameters with their individual types. The reason is that the algebraic notation builds the foundation for TCB functions where we aim at an implementation-oriented function signature with typed input parameters. For example, Section 3.4.5 has described a primitive action called *assignUserToRoles*, which assigns a set of roles to a user by modifying the state component UA_q:

$$\mathit{assignUserToRoles} : Q \times X \to Q$$
$$\mathit{assignUserToRoles}(q, (x_u, \{x_{r_1}, \ldots, x_{r_n}\})) := (U, S, UA_q \cup \{(u, r_1), \ldots, (u, r_n)\}, \mathit{user}, \mathit{roles})$$

In order to include this primitive action in the operation set of *State*, the input vector X has to be replaced by the types of the specific input parameters. As indicated by the parameter's indexes, the types of the input parameters are a user $x_u \in U$ and a set of roles $\{x_{r_1}, \ldots, x_{r_n}\} \in 2^{\mathit{Roles}}$ such that this primitive action can be refined as $\mathit{assignUserToRoles} : Q \times U \times 2^{\mathit{Roles}} \to Q$. It now remains to replace the model notation by algebraic notation. That means, the parameters of a primitive action have to be described by sorts rather than model sets. The signature of *assignUserToRoles* is then described as shown in Table 4.7; the sort *State* is defined by the core algebra *State* and the sorts *User* and *Set*[*Roles*] are sorts defined by component algebras as described above. This results in that *Set*[*Roles*] must be imported into *State*, even though it is not a state component. Hence, the imported sorts of

State do not only depend on a model's state components but also on the input parameters of a model's primitive actions and clauses.

operations	$assignUserToRoles : State \times User \times Set[Role] \to State$
semantics	$assignUserToRoles(q : State, u : User, \{r_1, \dots, r_n\} : Set[Role]) :=$ $(S_U, S_S, r\_\cup(R_{UR}, \{(u, r_1), \dots, (u, r_n)\}), Map_{SU}, Map_{SS_R})$

Table 4.7: Example of Rewriting a Primitive Action

The next step is to define the semantics of all primitive actions and clauses in algebraic notation by replacing (i) all model components with their algebraic representatives and (ii) the model operators with the appropriate operations of the component algebras. Here, it is significant that the interpretation of the algebra operations equals the interpretation of the model's primitives actions and clauses. For the above example that means, the state components U_q, S_q, UA_q, $user_q$, and $roles_q$ are replaced by the base sets S_U, S_S, R_{UR}, Map_{SU}, Map_{SS_R} of the sorts *Set*[*User*], *Set*[*Session*], *Relation*[*User*, *Role*], *Mapping*[*Session*, *User*], and *Mapping*[*Session*, *Set*[*Role*]]. Moreover, the model operator $\cup$ for relations is replaced by the operation $r\_\cup()$ defined on *Relation*[*User*, *Role*], since it has the same interpretation.

As a result, the core algebra *State* defines an n-tuple that exactly represents the state of a core-based model. The algebra's operation set is defined by the model's primitive actions and those clauses that are specified on the model state.

The last step of algebra engineering is the specification of the core algebra *ExtensionVector*. This is done in analogy to *State*. Besides the sort *Bool*, the algebra *ExtensionVector* imports all sorts that represent static model components and combines them to an n-tuple as defined by the underlying model. Besides the constructor *createExtVec*() and selector operations, e.g., *getSetElement*(), the operations of the algebra *ExtensionVector* are specified by those clauses that are defined on static model components; the remaining clauses should already be included in the operation set of *State*. To include these clauses in *ExtensionVector*, we apply the same procedure as already demonstrated for the algebra *State*. First, the general input parameter vector needs to be dissolved and replaced by specific input parameters. Afterwards the clauses' semantics are defined in algebraic notation by replacing the model components with their algebraic representations and the model operators with the appropriate operations of the component algebras. Result is a collection of all static model components in algebraic notation as shown in Table 4.8; here, the imported sorts as well as the operations and their input parameters are also exemplary since they are model-specific.

import	$Bool \cong \{true, false\}$
	$Element \cong e :=$ element of a nonempty set of a security model
	$Domain \cong d :=$ element of a nonempty set of a security model
	$Target \cong t :=$ element of a nonempty set of a security model
	$\dots$
	$Set[Element] \cong S := \{s_1, \dots, s_n \mid n \in \mathbb{N}\}$
	$Relation[Domain, Target] \cong R \subseteq \{D \times T \mid R\,is\,finite\}$
	$Mapping[Domain, Target] \cong Map \subset \{D_1 \times T_2 \mid Map\,is\,finite\}$
	$Matrix[Row, Column, Target] \cong M \subset \{R \times C \times T_3 \mid M\,is\,finite\}$
	$\dots$
sorts	$ExtVec \cong E := (S, R, Map, M, \dots)$
operations	$createExtVec : Set[Element] \times Relation[Domain, Target] \times$

	$\quad Mapping[Domain, Target] \times Matrix[Row, Column, Target] \rightarrow ExtVec$
	$getSetElement :\rightarrow Set[Element]$
	$getRelationElement :\rightarrow Relation[Domain, Target]$
	$getMappingElement :\rightarrow Mapping[Domain, Target]$
	$getMatrixElement :\rightarrow Matrix[Row, Column, Target]$
	$\ldots$
	$clause_1 : Relation[Domain, Target] \rightarrow State$
	$\ldots$
	$clause_n : Domain \times Target \rightarrow State$
semantics	$creatExtVec(S, R, Map, M) := (S, R, Map, M)$
	$getSetElement() := S$
	$getRelationElement() := R$
	$getMappingElement() := Map$
	$getMatrixElement() := M$
	$\ldots$
	$clause_1(x_p)$
	$\ldots$
	$clause_n(x_p)$

Table 4.8: Abstract Core Algebra *ExtensionVector*

4.4.2.2 Abstract Data Type Engineering

The result of algebra engineering is an algebraic specification of a core-based model, consisting of a set of component and core algebras. Based on this set of algebras, ADT engineering now derives the TCB functions that are necessary to implement the underlying model and that hence are contained in the functional component `security model functions` of a TCB's policy-dependent RTE (Figure 4.17). Depending on a specific core specialization, these functions support the model components of the automaton, including the state q and the transition function δ, as well as the extension vector E. Additionally, they implement the underlying mathematical base types such as sets and relations, in dependence on a model's components and their types. In the following, we discuss how TCB functions are derived from the set of algebras.

The goal of ADT engineering is to provide the basis for a formal rule system and an automated TCB composition tool. Therefore, ADT engineering must also have a sound formal foundation. To this end, ADT engineering establishes a mapping (called φ in the following) that maps an algebra operation to a set of ADT operators that is required to implement the operation. This is possible due to the causal dependencies between security policies (represented by a set of algebras) and TCB functions (represented by ADT operators). φ is then controlled by a rule system that formalizes causal dependencies between policies and TCB functions (Section 4.4.4). The semantics of the ADT operators are described by operations of a set-theoretic universe (formalized by an additional mapping β); only in case ADT operators cannot be mapped to operations in the universe, axioms are applied.

Algebraic specifications are based on a set-theoretic universe. In the context of TCB engineering, the set-theoretic universe $\mathcal{U}$ is based on urelements E, e.g., users, roles, or objects, containing sets of urelements $S(E)$, sets of all finite subsets of urelements $\mathcal{F}(S(E))$, and ordered pairs ($S(E) \times S(E)$, $S(E) \times S(E) \times S(E)$). Formally, the universe is defined as $\mathcal{U} = \bigcup_{n \geqslant 0} S^n(E)$ where $S(X) = X \cup \mathcal{F}(X) \cup X \times X \cup X \times X \times X$. The universe operation set $\mathcal{U}_O$ consists of all set-theoretic operations defined on elements of $\mathcal{U}$.

Every time when algebras are engineered for a specific model as shown in Section 4.4.2.1, a mapping $\alpha : \mathcal{O}_A \rightarrow \mathcal{U}_O$ from the set of algebra operations $\mathcal{O}_A$ to the universe operation set $\mathcal{U}_O$ is established. That means, an algebra operation o_A that is mapped to a universe operation u_O has identical semantics as this set-theoretic operation, since the semantics of the algebra operations is defined by the semantics of the universe operation. For example, the operation $s\_\cup()$ of the abstract component algebra *Set* (Table 4.1) merely is a rewriting of the standard set operation $\cup$, since $\alpha(s\_\cup) := \cup$.

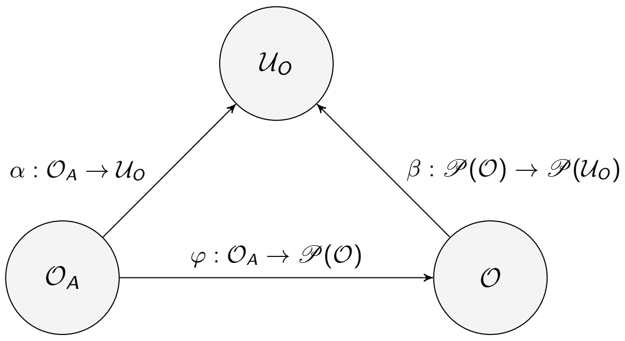
$$\begin{tikzcd} & \mathcal{U}_O & \\ \mathcal{O}_A \arrow[ur, "\alpha : \mathcal{O}_A \rightarrow \mathcal{U}_O"] \arrow[rr, "\varphi : \mathcal{O}_A \rightarrow \mathscr{P}(\mathcal{O})"] & & \mathcal{O} \arrow[ul, "\beta : \mathscr{P}(\mathcal{O}) \rightarrow \mathscr{P}(\mathcal{U}_O)"'] \end{tikzcd}$$

Figure 4.19: Mappings of Engineering a Policy-dependent RTE

In the course of ADT engineering two additional mappings are defined as shown in Figure 4.19. ADTs are derived from the operations of a model's algebras by a rule system based on causal dependencies between security policies and TCB functions. In this process the operations do not undergo a semantic change; the semantics of the ADTs' interface operators exactly equals the semantics of the algebras' operations, and a set of basic ADT operators implements the interface operator. The rule system describes the mapping $\varphi : \mathcal{O}_A \rightarrow \mathscr{P}(\mathcal{O})$ where $\mathcal{O}_A = \mathcal{O}_{A_S} \cup \mathcal{O}_{A_R} \cup \mathcal{O}_{A_{Map}} \cup \mathcal{O}_{A_M}$ are the algebra operations and $\mathcal{O} = \bigcup_{i=1}^{n} \mathcal{O}_i$ are the basic operator sets of the ADTs $\Sigma_i = (\mathcal{S}_i, \mathcal{O}_i)$, which then semantically equal the algebra operation. The set of interface operations is not derived by the rule system since interface operations do not provide additional TCB functionality; they only wrap the basic ADT operators that implement the corresponding algebra operation. However, interface operators are added to $\mathcal{O}$ since they specify the interface functions of the `security model functions` component, contribute to a concise ADT specifications, and allow for structuring the derived TCB functions.

In order to describe the interpretation of ADT operators, we define a second mapping $\beta : \mathscr{P}(\mathcal{O}) \rightarrow \mathscr{P}(\mathcal{U}_O)$, which maps a set of ADT operators to a set of universe operations. That means, a set of ADT operators that is mapped to a set of universe operations implements the semantics of all these universe operations. At the same time, the ADT's interface operator is also mapped to these universe operations, because it shares the same semantics.

This approach is feasible since the operations of $\mathcal{U}_O$ are well-defined due to the design criteria of $\mathcal{U}$. The advantage of this approach is that we do not have to define axioms for ADT operators, whose precise interpretation is already given by a model and hence its algebra. We only have to define axioms for those ADTs, which cannot be mapped to universe operations, because they are not based on set theory. On the whole, the efforts for ADT engineering are greatly decreased, since we only have to define axioms and deal with their consistency and completeness for a few ADTs; many of the derived ADT operators can be mapped to universe operations.

In the following, we demonstrate the ADT engineering approach by means of the operation

$s\_\cup \in \mathcal{O}_{A_S}$ of the algebra *Set*[*Element*]; the approach is analogous for all other algebras and their operations. As mentioned above, the algebra operation $s\_\cup()$ shares the same semantics as the standard union operation $\cup \in \mathcal{U}_O$; thus, $\alpha(s\_\cup) := \cup$. Before deriving the TCB functions that are necessary to implement this algebra operation, the model algebra first requires an ADT `Set[Element]` with an equivalent interface operator, called `s_union`(). Since this interface operator has the same semantics as $s\_\cup \in \mathcal{O}_{A_S}$ and thus as the universe operation $\cup \in \mathcal{U}_O$, we map $\beta(\{s\_union\}) := \{\cup\}$. The next step is to derive the basic ADT operators that implement this interface operator:

$\varphi(s\_\cup) := \{$
```
Elem_T next(Cont_T c),
Set_T createSet(),
Elem_T create(),
Set_T s_insert(Set_T s, Elem_T e),
Bool_T equals(Elem_T e1, Elem_T e2),
Cont_T insert(Cont_T c, Elem_T e),
Bool_T false(),
Bool_T true()}
```

As can be seen, six TCB functions (basic ADT operators) defined on three additional sorts (`Cont_T`, `Elem_T`, and `Bool_T`) are required to implement the interface function `s_union`(). The functions `createSet()` and `s_insert`() are additional interface functions of `Set_T`, which are required to construct the input parameters of `s_union`(), and `create()` is exclusively required to construct the input parameter of `s_insert`(). More precisely, `Cont_T` is a basic container sort that allows for storing varying numbers of elements, which are represented by the parameter sort `Elem_T`, and `Bool_T` represents the algebra *Bool*. The ADT `Set[Element]` (short `Set_T`) encapsulates the container sort and the interface operators of `Set_T` are responsible for implementing set properties. In the context of unifying sets according to the semantics of $\cup$ this means, the resulting set must not contain redundant elements. Thus, an iterator function `next()` is required, which returns the next elements of a container to be compared to an element of the second container by the function `equals()`. The latter returns a Boolean expression, which is constructed either by `false()` or `true()`, depending on the result of the comparison. Only if an element is not already contained by the set, it will be inserted in the container by `insert()`. Thus, the combination of these functions implements the interface operator `s_union`() of `Set_T` such that $\beta(\{\mathit{next}, \mathit{equals}, \mathit{insert}, \mathit{false}, \mathit{true}\}) := \{\cup\}$. The functional redundancy of such a set of ADT operators is ensured by the operators' names. An operator can only be added if there is no other operator with the same name, independent of its signature. Note that we do not consider sequence control mechanisms. Such mechanisms are provided by implementation-specific platforms and can be used to implement the derived TCB functions.

It remains to discuss some characteristics of the mappings involved in TCB engineering. As long as the algebraic foundation for algebra engineering is not modified, i.e., enhanced by a novel component algebra, the set-theoretic universe is defined identically for each core-based model. This results in that the mapping $\alpha : \mathcal{O}_A \to \mathcal{U}_O$ from algebraic operations to universe operations is not surjective for the majority of models. However, depending on the primitive actions and conditions used by a model, it may happen that α becomes surjective. Additionally, α generally is not injective. The reason is that due to the design of the algebra foundation, there are usually at least two operations that share the same semantics. For example, $s\_\varnothing()$ and $r\_\varnothing()$ both are mapped to the universe operation $\varnothing$; the same holds for $s\_insert()$ and $s\_\cup()$, which are mapped to $\cup$. However, these algebra operations may differ in their number and sorts of input parameters as well as in their output parameter sort. Result is that different TCB functions must be derived.

The mapping $\beta : \mathscr{P}(\mathcal{O}) \to \mathscr{P}(\mathcal{U}_O)$ maps ADT operators that are not based on universe operators to the empty set, e.g., constructors for `Cont_T` and `Elem_T` where we have to define the ADT operators' semantics by a set of axioms. Therefore, β is not injective. In the majority of cases, β is not surjective, since $\mathcal{U}_O$ and thus its power set may contain universe operations that are not required for the underlying model's algebras and thus ADT operators. Here, the argumentation for the characteristics of α also applies.

The rule system formalized by the mapping $\varphi : \mathcal{O}_A \to \mathscr{P}(\mathcal{O})$ from algebra operations to ADT operators representing TCB functions is not surjective and not injective. The reasons are as follows. φ is not surjective since there are TCB function sets, e.g., some singletons, that do not have a corresponding element in the set of algebra operations. φ is not injective, because it may happen that at least two algebra operations require the same set of TCB functions for their implementation (Section 4.4.4).

The last step of ADT engineering is to define the ADTs `State` and `ExtensionVector` for the algebras *State* and *ExtensionVector*. Here, we do not derive additional TCB functions but merely add interface functions according to the algebras. These interface functions are implemented by those ADT operators that are already contained in $\mathcal{O}$. An example is given in Section 4.5.1.

Result is a set of ADTs whose operators are defined by the elements of $\mathcal{O}$ and which represent a TCB's functional range contained in the component `security model functions` of the policy-dependent RTE. It now remains to specify the last component of the policy-dependent RTE, called the `interceptor`. This is done in the following section.

4.4.3 Abstract Model Instance Functionality

The goal of this section is to illustrate the model instance functions of a TCB's policy-depend RTE. These functions are concentrated in the functional component `interceptor`; so this section discusses the `interceptor` and its interrelations with `executable security policies`.

The functional range of the `interceptor` is not defined by a TCB's policies although it is policy-specific. In fact, the interface functions of the `interceptor` are defined by the OS and its applications that the policies are responsible to protect. If the authorization scheme of a policy does not match the interface of the `interceptor`, the policy is not qualified to protect the given OS.

The reason for such a design is a combination of two requirements: substitutionality of security policies and multi-policy support (Section 4.1.2). Substitutionality of policies requires that these can be easily integrated, updated, and removed. This implies that the interfaces of the `executable security policies` to the `specific object managers` are to be modified very rarely. However, in case a modification is required, the administrator would have to check every `specific object manager` that calls a policy. In the long run, this might interfere with the requirement of total-mediation of a policy (reference monitor principles [17, 97, 125]). On top of that, multi-policy TCBs aggravate this situation since in the worst case all `object managers` have to call every single policy that participates in policy decision making. In both cases updating a policy or replacing a policy by multiple ones is complex and error-prone.

For this reason, the policy-depend RTE contains the functional component `interceptor`, which basically acts as a proxy by wrapping all commands of the policies' authorization schemes. The `specific object managers` then call the `interceptor` which maps the policy request to that policy that implements the command of the request. By this means, whenever a policy is updated or replaced, any necessary modification only concerns the `interceptor` and not the set of `specific object managers`. In case of multi-policy TCBs, only the `interceptor` has to know which policy to call for processing a request. As a consequence, policies can be integrated, updated, and removed more easily.

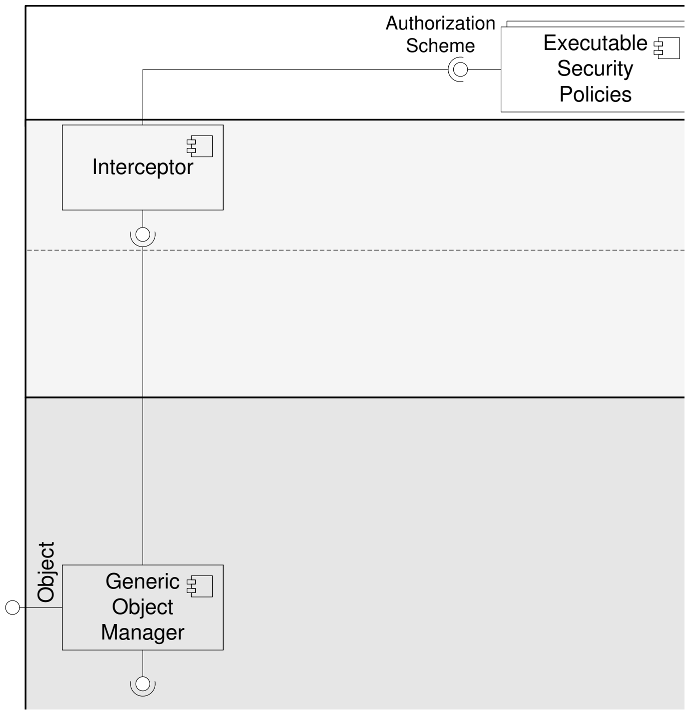

Figure 4.20: Interceptor

As shown in Figure 4.20, the `interceptor` provides an interface to the `generic object manager` and uses the interfaces of all `executable security policies`. The interface to the `generic object manager` is defined by the OS and its applications. In a single-policy TCB, the `executable policy` has to implement the entire interface; in a multi-policy TCB the interface can be shared by many `policies` with or without overlapping authorization schemes[6]. In either case, the `interceptor` implements a mapping from its interface functions to the commands of the policies' authorization schemes. This means, the set of interface functions $\mathcal{C} = \bigcup_{i=1}^{n} C_i$ that is provided by the `interceptor` to the `generic object manager` is composed by all command sets C_i of the policies integrated in a TCB, where each set contains state-modifying and non-state-modifying commands (Section 4.1.1). $\mathcal{C}$ must then equal the required interface functions of a given OS.

```
public Bool_T c_1(Set_T s, Elem_T e)
public Bool_T c_2(Rel_T r, Dom_T d)
...
public void c_m(Dom_T d, Tar_T t)
```

[6]The scope of this dissertation are multi-policy TCBs that execute independent `policies` without overlapping authorization schemes.

As shown above, the `interceptor` contains m commands with $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$, whose input parameters are defined by command-specific input vectors $x \in X$ with $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$, and where $T = \{D_1, ..., D_n, d_{1_q}, ..., d_{n_q}, e_1, ..., e_h, 2^{e_1}, ..., 2^{e_h}\}$ contains all components of a model's state set, state, and extension vector, along with the power sets of the extension vector components (Section 3.2). Here, the input parameters are merely examples; we discuss how to derive command-specific input parameters in the following.

Section 4.4.2 has illustrated an engineering method based on model algebras that basically reflect the mathematical types of model components and ADTs that are derived from these model algebras. This method also has an effect on the signatures of the functions within the `interceptor`. Besides additional ADTs such as `Bool_T` or `Elem_T`, we have derived an ADT for each component algebra, which are now used as input parameter types for the interface functions. For example, the command *assignRoles* of the RBAC HIS policy (Appendix B.2) is rewritten as interface function as follows.

```
public void assignRole(Session_T s, User_T u, Role_T r).
```

As already explained in Section 4.4.2, the input parameter vector X must be dissolved to individual input parameter types. Here, it is dissolved to $X = S \times U \times R$. Applying the algebra engineering method, an element of S, U, or R is represented by the sort `Element`, from which specialized ADTs like `Session_T`, `User_T`, or `Role_T` can be derived (Section 4.4.4). Note that this function does not have an output parameter, because *assignRoles*() is a state-modifying command. While state-modifying commands generally do not have a return value, non-state modifying commands have return values of type *Bool*, using the ADT `Bool_T`. For examples refer to Section 4.5.2, which presents the functional range of the `interceptor` for a TCB that enforces the RBAC HIS policy.

It now remains to formally define the causal dependencies used to derive TCB functions from the algebraic notation of core-based models.

4.4.4 Causal Dependencies

Engineering a TCB's policy-dependent RTE requires tool support that allows for generating and composing the policy-dependent functions of a TCB. Both algebra and ADT engineering set the course for tool support by providing a theoretical foundation that is now used to formalize causal dependencies between security models and TCB functions. Due to the applied formalisms, this rule system can be integrated in an automated tool that composes the functional perimeter of a TCB's policy-dependent RTE. With respect to the requirements (Section 4.1.2), it is important that the resulting functional perimeter does not contain redundant functions and is complete with respect to the policies.

The goal of this section is to define the formal rule system. The idea is that any rule of the rule system leads to a set of TCB functions as a part of a TCB's implementation. In the following, we present the formal rules for the algebras *Set*[*Element*] and *Relation*[*Domain*, *Target*] and illustrate the resulting ADTs and their interrelations. The rules for the remaining algebras *Mapping*[*Domain*, *Target*] and *Matrix*[*Row*, *Column*, *Target*] are illustrated in Appendix C due the same design principles.

Input of the rule system is a set of component algebras that describe the components of a core-based model; output is a set of ADT operators that are the functions contained by the `security model functions` component of the policy-dependent RTE. The formal rule system then defines the mapping $\varphi : \mathcal{O}_A \rightarrow \mathscr{P}(\mathcal{O})$, where $\mathcal{O}_A = \mathcal{O}_{A_S} \cup \mathcal{O}_{A_R} \cup \mathcal{O}_{A_{Map}} \cup \mathcal{O}_{A_M}$ contains the operations of all component algebras and $\mathcal{O} = \bigcup_{i=1}^{n} \mathcal{O}_i$ is a set of ADT operators defined by the operation sets of all derived ADTs with signature $\Sigma_i = (\mathcal{S}_i, \mathcal{O}_i)$. The rule system defines a set of logic implications $A \rightarrow B$, where A is a statement containing necessary and sufficient conditions for the existence of TCB functions and B is a statement about the functions contained in a TCB. Conditions are denoted in first-order logic; they are defined on the component algebra's operations and on the mapping $\alpha : \mathcal{O}_A \rightarrow \mathcal{U}_O$. B defines the mapping $\varphi : \mathcal{O}_A \rightarrow \mathscr{P}(\mathcal{O})$.

In general, the formal rule system is geared to the abstract component algebras of the algebra foundation defined in Section 4.4.2. That means, the rule system provides a set of predefined functions for composing a policy-dependent RTE. In case the algebra foundation is not sufficient due to future model trends, both the algebras' operation sets as well as the formal rule system need to be extended. Since the algebra operations are based on the set-theoretic universe, the latter may be extended, too.

4.4.4.1 Set

In the following we define formal rules for the algebra *Set*. For this purpose, we need two predicates to evaluate the input and output sorts of the algebra's operations.

1. $param\_sort(x, A)$: x is a parameter sort of algebra A
2. $set\_sort(x)$: x is of sort *Set*

The algebra $Set[Element]$ produces the ADT `Set[Element]` (short `Set_T`) that contains a set of interface operators, which is defined by the ADT's operator set and equals the operations of the algebra *Set*. The operators of `Set_T` are implemented by basic operators that are derived as shown below.

The first rule focuses on a constructor operation of the algebra, which is exemplarily called $s\_\varnothing()$ in Table 4.1 and is semantically equal to $\varnothing \in \mathcal{U}_O$, resulting in $\alpha(s\_\varnothing) := \varnothing$. The rule states that if an algebra operation o_A semantically equals $\varnothing$, it produces a TCB function called `createContainer()` defined on a basic ADT `Container[Element]` (short `Cont_T`):

1. Condition	TCB Functions and Mapping to Universe Operations
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \varnothing$	$\varphi(o_A) := \{$ `Cont_T createContainer()` $\}$ $\beta(\{createContainer\}) := \{\varnothing\}$ $\beta(\{createSet\}) := \{\varnothing\}$ (Interface Function)

`Cont_T` represents a container data type for elements of the same type, implementing the ADTs `Set_T`, `Rel_T`, `Map_T`, and `Matrix_T`. In the context of an implementation-specific platform, `Cont_T` may be implemented by any given specific data type such as lists, arrays, or vectors; specific properties of the ADTs, e.g., nonredundancy of set elements, are implemented on top of `Cont_T` by the operators of `Set_T`, `Rel_T`, `Map_T`, and `Matrix_T`. `createContainer()` implements $s\_\varnothing()$ and is hence semantically equal to $\varnothing \in \mathcal{U}_O$ such that $\beta(\{createContainer\}) := \varnothing$. On top of that, $s\_\varnothing()$ becomes an interface function of the ADT `Set_T` and is called `createSet()` in the following, since this is more implementation-oriented. Additionally, it is also mapped to $\varnothing \in \mathcal{U}_O$ since it is implemented by `createContainer()`.

The second rule is concerned with an operation that inserts a single element in a set (called *s_insert*() in Table 4.1). Such an operation basically equals the set operation $\cup$; however, it requires an additional TCB function to create the element to be inserted and must hence be distinguished. The condition then requires that the operation is semantically equal to $\cup$, must have exactly one input parameter $s_{i,A}$ that is a parameter sort of the algebra *Set*, and its output parameter s_A must be of sort *Set*. The latter condition part is necessary, since the algebras *Relation*, *Mapping*, and *Matrix* have identical operations but require different ADT operators.

If such an operation is contained in $\mathcal{O}_A$, a TCB requires six functions for its implementation: The ADT `Cont_T` must implement an operator `next()` that iterates the container and returns the next element to be compared to the one to be inserted. Here, `Elem_T` is the basic type of container elements, which is provided by an ADT called `Element`. The latter must implement a constructing operator `create()` and a comparing operator `equals()`. Again, to implement the comparing operator, an ADT `Bool_T` is required, providing the constructors `false()` and `true()`, which allow for evaluating the results of comparing two container elements. Finally, `insert()` allows to add the newly created element to the container. Besides `create()`, the set of the derived ADT operators is encapsulated by the interface function `s_insert`(), which therefore maps to the same universe operator $\cup \in \mathcal{U}_O$. The remaining function `create`() is required to construct an object of type `Elem_T` as input parameter for `s_insert`().

2. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$
$\wedge\ \exists! s_{i,A} : \mathit{param\_sort}(s_{i,A}, A_S)$
$\wedge\ \mathit{set\_sort}(s_A)$
	$\varphi(o_A) :=$ { `Elem_T next(Cont_T c),`
	`Elem_T create(),`
	`Cont_T insert(Cont_T c, Elem_T e),`
	`Bool_T equals(Elem_T e`$_1$`, Elem_T e`$_2$`),`
	`Bool_T false(),`
	`Bool_T true()`}
	$\beta(\{\mathit{next}, \mathit{insert}, \mathit{equals}, \mathit{false}, \mathit{true}\}) := \{\cup\}$
	$\beta(\{\mathit{s\_insert}\}) := \{\cup\}$ (Interface Function)

The next rule is defined on an algebra operation that checks if an element belongs to a given set and semantically equals $\in$. If such an operation exists and has exactly one input parameter that is of sort *Set* and exactly one other input parameter that is a parameter sort, five functions are required to implement this operation. Again, we must distinguish the sorts of the input parameters since the other algebras of the algebra foundation have similar functions that equal $\in$ and have the same output parameter sort *Bool*, but require different TCB functions. The resulting set of TCB functions for this algebra operation is a subset of the required TCB functions that implement *s_insert*(). Hence, we do not discuss their functionality again. The only difference is that the ADT `Set_T` must now implement the interface function `s_isElement`() that encapsulates four of these five functions. Again, the remaining function `create`() is required to construct an object of type `Elem_T` as input parameter for `s_isElement`().

3. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \in$
$\wedge\ \exists! s_{i,A} : \mathit{set\_sort}(s_{i,A})$
$\wedge\ \exists! s_{j,A} : \mathit{param\_sort}(s_{i,A}, A_S)$
$\wedge\ i \neq j$
	$\varphi(o_A) :=$ { `Elem_T next(Cont_T c),`
	`Elem_T create(),`
	`Bool_T equals(Elem_T e`$_1$`, Elem_T e`$_2$`)`
	`Bool_T false(),`

	`Bool_T true()`}
	$\beta(\{next, equals, false, true\}) := \{\in\}$
	$\beta(\{s\_isElement\}) := \{\in\}$ (Interface Function)

A generalization of the algebra operation *s_insert*() is an operation that inserts multiple elements instead of a single one into a given set, called $s\_ \cup ()$ in Table 4.1. For this reason, the operation semantically equals *s_insert*() but differs in its input parameter sorts. While the rule for *s_insert*() requires exactly one input parameter to be a parameter sort, all input parameters of $s\_ \cup ()$ must be of sort *Set*. This condition part is also required to distinguish this operation from similar operations of the other component algebras, e.g., $r\_ \cup ()$ and $mp\_ \cup ()$. Result a set of six basic ADT operators equally to the required ADT operators of *s_insert*(). The only difference is that in addition to the constructor operator for objects of type `Elem_T`, we also require the set constructor for creating the first input parameter; these functions are similar to the ones derived from $s\_\varnothing()$. For this reason, the required operators equals the ones derived from the 1. and the 2. rule so that these rules can be reused. Besides `createSet`() and `s_insert`(), `Set_T` must now contain the interface operator `s_union`(). The basic operators for implementing `s_union`() are defined by a unified set of the derived operator sets of the 1. and 2. rule.

4. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \{\cup\}$ $\wedge \forall s_{i,A} : set\_sort(s_{i,A})$	$\varphi(o_A) :=$ { `Set_T createSet(),` `Set_T s_insert(Set_T s, Elem_T e)`} $\beta(\{s\_union\}) := \{\cup\}$ (Interface Function)

In analogy to the 4. rule, there needs to be a rule for an operation that removes multiple elements from a set. The only difference in the condition is that the algebra operation must be equal to $\setminus \in \mathcal{U}_O$ rather than $\cup \in \mathcal{U}_O$. The required sets of TCB functions are also equal except for an additional function: `Cont_T` must now implement an deleting operator, called `delete()`. We call the corresponding interface operator `s_diff()`. The operators `createSet()`, `create()`, and `s_insert()` are required to create the input parameters of `s_diff()`.

5. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \{\setminus\}$
$\wedge \forall s_{i,A} : set\_sort(s_{i,A})$
	$\varphi(o_A) :=$ { `Elem_T next(Cont_T c),`
	`Set_T createSet(),`
	`Elem_T create(),`
	`Bool_T equals(Elem_T e`$_1$`, Elem_T e`$_2$`),`
	`Cont_T delete(Cont_T c, Elem_T e),`
	`Set_T s_insert(Set_T c, Elem_T e),`
	`Bool_T false(),`
	`Bool_T true()`}
	$\beta(\{next, equals, delete, false, true\}) := \{\setminus\}$
	$\beta(\{s\_diff\}) := \{\setminus\}$ (Interface Function)

The last rule for the algebra *Set* is concerned with an operation that determines the number of elements of a given set. To distinguish this operation from other algebra operations, the condition checks whether it semantically equals the universe operation $|X| \in \mathcal{U}_O$ (cardinal number) and has exactly one input parameter of sort set. The set of TCB functions required to implement this operation consists of three elements: `next()` iterates objects of `Cont_T` and returns the next element, `create()` constructs an object of type `Nat_T`, which is an ADT representing natural numbers, and `add()` defined on `Nat_T` allows to add a natural number to a given one. An additional interface operator of `Set_T` wraps these ADT operators and is called `s_numOfElements()`.

6. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \{\|X\|\}$
 $\wedge\ \exists!\, s_{i,A} : set\_sort(s_{i,A})$
	$\varphi(o_A) :=$ { `Elem_T next(Cont_T c),`
	 `Nat_T create(),`
	 `Nat_T add(Nat_T n, Nat_T m),`
	 $\beta(\{next, create, add\}) := \{\|X\|\}$
	 $\beta(\{s\_numElements\}) := \{\|X\|\}$ (Interface Function)

Applying these rules to a model-specific component algebra *Set* results in a set of ADTs whose number of elements and operators naturally depend on a given security model. Assuming that an algebra defines all operations for which we have specified formal rules, the resulting TCB must implement all named ADTs. Otherwise, it must implement at least the ADTs `Set_T`, `Elem_T`, and `Cont_T`, and probably `Bool_T` since almost all algebra operations require logical values. Only if an algebra operation exists that is based on the cardinal number, the ADT `Nat_T` is required in a TCB. We present all ADTs in the following; the notation of the ADTs is roughly based on [75].

`Set[Element]` is a parameterized ADT that defines the sort `Set_T` by a total of six ADT operators (Table 4.9). It imports the sorts `Bool_T`, `Elem_T`, and `Nat_T`, since these are required to define the sorts of the operators' input and output parameters. In general, the set of operators must be equal to the operation set of the according component algebra, which here is *Set*. At the same time, the operations' parameter sorts are replaced by the corresponding ADT sorts. All ADT operators of `Set_T` are interface functions, which are implemented by basic ADT operators. Since their semantics is defined by the mapping β, we do not need to define axioms for these operators.

sorts	`Set_T`
import	`Bool_T`, `Elem_T`, `Nat_T`

operators	`createSet:` $\rightarrow$ `Set_T`
	 `s_insert: Set_T` $\times$ `Elem_T` $\rightarrow$ `Set_T`
	 `s_isElement: Set_T` $\times$ `Elem_T` $\rightarrow$ `Bool_T`
	 `s_union: Set_T` $\times$ `Set_T` $\rightarrow$ `Set_T`
	 `s_diff: Set_T` $\times$ `Set_T` $\rightarrow$ `Set_T`
	 `s_numElements: Set_T` $\rightarrow$ `Nat_T`

Table 4.9: ADT `Set[Element]` (`Set_T`)

After having subsumed the operators of the ADT `Set_T`, we now specify the ADTs of the imported sorts. The ADT `Bool_T` (Table 4.10) represents Boolean values. As defined by the formal rule system, `Bool_T` must provide two operators `true`() and `false`(), which are constructor functions to create any object of sort `Bool_T`. Since the semantics of these operators is not defined by universe operators, we have to define a set of axioms that describe their interpretation.

sorts	`Bool_T`

operators	`true: → Bool_T`
	`false: → Bool_T`

axioms	`true() = true`
	`false() = false`

Table 4.10: ADT `Bool_T`

`Elem_T` is the parameter sort of `Set_T` and is defined by an additional ADT `Element` as shown by Table 4.11. Due to the formal rule system, `Elem_T` defines two operators: `create`() as constructing operator and `equals`() to compare two `Elem_T` objects. Again, the interpretation of these functions is not described by universe operators so that we have to provide a set of axioms, which is based on [75].

sorts	`Elem_T`
import	`Bool_T`

operators	`create: → Elem_T`
	`equals: Elem_T × Elem_T → Bool_T`

variables	$\forall e_1, e_2, e_3 :$ `Elem_T`

axioms	equals(e_1,e_1) = true()
	equals(e_1,e_2) → equals(e_2,e_1) = true()
	(equals(e_1,e_2) ∧ equals(e_2,e_3)) → equals(e_1,e_3) = true()

Table 4.11: ADT Element (`Elem_T`)

It remains to define the ADTs `Cont_T` and `Nat_T`. As mentioned above, `Cont_T` represents a container data type storing an arbitrary number of element objects to implement the ADTs representing the component algebras (Table 4.12). It provides four operators that allow to create container objects (`createContainer()`), to iterate a container (`next()`), and to insert and delete container elements (`insert()`, `delete()`). In analogy to the ADTs `Bool_T` and `Elem_T`, we have to define a set of axioms to specify the operators' semantics.

sorts	`Cont_T`
import	`Elem_T`

operators	`createContainer: → Cont_T`
	`next: Cont_T → Elem_T`
	`insert: Cont_T × Elem_T → Cont_T`
	`delete: Cont_T × Elem_T → Cont_T`

variables	$\forall e :$ `Elem_T`, $C :$ `Cont_T`

axioms	`delete(insert(C,e),e) = C`
	`delete(createContainer(),e) = createContainer()`
	`delete(insert(createContainer(),e)e) = createContainer()`

Table 4.12: ADT Container (`Cont_T`)

As mentioned above, the ADT `Nat_T` (Table 4.13) represents the set of natural numbers. Due to the above defined causal dependencies, it only provides two operators: an constructor `createNat()` to create arbitrary natural numbers and an adding operator `add()`, which adds two arbitrary numbers. Again, we have to define axioms to specify their semantics. In this context, we have to define an additional ADT operator `succ()` for a sound ADT specification. However, this function is not derived from the algebra operations such that it does not belong to the functional perimeter of a TCB.

sorts	`Nat_T`

operators	`create:` $\rightarrow$ `Nat_T`
	`add: Nat_T` $\times$ `Nat_T` $\rightarrow$ `Nat_T`
	`(succ: Nat_T` $\rightarrow$ `Nat_T)`

variables	$\forall$`n,m : Nat_T`

axioms	`add(create(),n) = n`
	`add(n,succ(m)) = succ(add(n,m))`

Table 4.13: ADT `Nat_T`

All derived ADTs interrelate with each other by importing specific ADTs as sorts; Figure 4.21 gives an overview of the ADTs' interrelations by means of an UML class diagram. Here, we do not aim at an object-oriented design of the TCB; however, class diagrams provide the best means to illustrate the interrelations of the ADTs in detail. Every ADT is denoted by a class and named by the sort it provides. As can be seen, the abstract component algebra *Set* requires a total of five ADTs for its implementation. `Set_T` contains a single attribute of type `Cont_T`, implementing the underlying container data type; every object of type `Set_T` existentially depends on exactly one `Cont_T` object. On top of that, `Set_T` uses the functions of the `Nat_T` class to implement its function `numElements()`. Every `Cont_T` object is associated with an arbitrary number of `Elem_T` objects. The latter uses the interface functions of the `Bool_T` class for implementing its comparing function `equals()`. `Elem_T` is defined as an abstract class with two abstract functions (denoted by italic style). This is not required for the TCB functions implementing the algebra *Set*, but for all other component algebras.

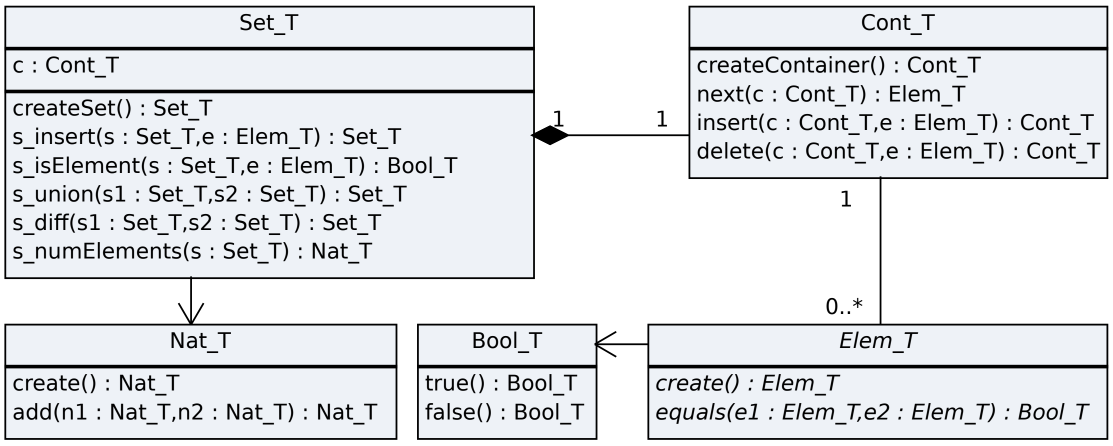
Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Set_T,e : Elem_T) : Set_T
s_isElement(s : Set_T,e : Elem_T) : Bool_T
s_union(s1 : Set_T,s2 : Set_T) : Set_T
s_diff(s1 : Set_T,s2 : Set_T) : Set_T
s_numElements(s : Set_T) : Nat_T
Cont_T
createContainer() : Cont_T
next(c : Cont_T) : Elem_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
1
1
1
0..*
Nat_T
create() : Nat_T
add(n1 : Nat_T,n2 : Nat_T) : Nat_T
Bool_T
true() : Bool_T
false() : Bool_T
Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T

Figure 4.21: ADTs Derived from $Set[Element]$

In the following we enhance the formal rule system by rules for the algebra *Relation*. Here, we apply the same approach; we present the formal rules before we illustrate the resulting ADTs and their interrelations. As already mentioned, the rules for the remaining algebras *Mapping* and *Matrix* can be found in Appendix C.

4.4.4.2 Relation

Since relations are sets of ordered pairs, the operation set of the algebra *Set* naturally is a subset of the operation set of *Relation*. This results in that the set of TCB functions for implementing *Set* also is a subset of the TCB function set implementing *Relation*. However, a TCB requires additional functions to support ordered pairs of elements rather than primitive elements. This affects the rule system as follows: Even though the conditions are similar to those defined on *Set*, they lead to different, i.e., larger, sets of TCB functions. On top of that, core-based models apply relation-specific operations, e.g., $\lhd$ or $\rhd$, which have to be considered by additional rules.

To define formal rules for the algebra *Relation*, an additional predicate is required:

3. *relation_sort*(x): x is of sort *Relation*

In analogy to `Set_T`, the algebra *Relation* produces an ADT `Relation[Element]` (short `Rel_T`), which defines a set of interface operators that equals the operation set of *Relation* and is implemented by operators of the derived ADTs. The first rule concerning the constructing operation of *Relation* is identical to the first rule of *Set*. The reason is twofold: (i) Both constructor operations semantically equal $\varnothing \in \mathcal{U}_O$. (ii) The ADTs `Set_T` as well as `Rel_T` are both implemented by the container data type `Cont_T`, thus resulting in the identical set of ADT operators.

1. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \varnothing$	$\varphi(o_A) :=$ { `Cont_T createContainer()`} $\beta(\{\mathit{createContainer}\}) := \{\varnothing\}$ $\beta(\{\mathit{createRelation}\}) := \{\varnothing\}$ (Interface Function)

A small difference lies in the ADTs though; instead of `Set_T`, a TCB now contains `Rel_T`, providing the interface function `createRelation()` to match the component algebra *Relation*. As mentioned earlier, neither `Set_T` nor `Rel_T` and their operators add new functionality to a TCB so that the rule system does not consider these functions. Hence, the rules concerning the constructing operation of *Set* and *Relation* are identical and can be subsumed to one rule: the condition parts of both rules are subsumed to one condition: $\exists o_A \in \mathcal{O}_A, : \ \alpha(o_A) = \varnothing$, resulting in $\varphi(o_A) :=$ { `Cont_T createContainer()`}. This also holds for the algebras *Mapping* and *Matrix*.

While the ADTs derived from *Set* and *Relation* have many similarities, the remarkable difference lies in the specification of the ADT `Elem_T`. The latter now represents an ordered pair consisting of basic elements of types `Domain` and `Target` instead of a single element. That means, two new ADTs called `Dom_T` and `Tar_T` are required, representing the basic elements of an ordered pair. Note that the ADTs `Elem_T`, `Dom_T`, and `Tar_T` along with their operators are identical. We avoid functional redundancy by means of the same operator names.

The 2. rule is geared towards an algebra operation that inserts an element into a given relation as part of the algebra's object construction (compare 2. rule of *Set*). To distinguish this operation from a similar operation, called *mp_insert*() of algebra *Mapping* (Table 4.4), requiring different TCB functions due to its left-total and right-unique properties, the condition has to check whether the operation semantically equals $\cup \in \mathcal{U}_O$, has at least one input parameter which belongs to a parameter sort (here *Domain* or *Target*), has exactly one input parameter of sort *Relation*, and its output parameter is of sort *Relation*. If such an operation belongs to *Relation*, a TCB needs to implement a total of eight functions. First, the element to be inserted must be created by `create()`. This implies that the domain and the target elements are created before, which is also done by `create()` but returning different types. Since the semantics of all `create()`-functions is the same, i.e., creating an object, `create()` is only added once to the function set. Once the tuple element has been created, the container must be iterated by `next()` to check whether an element with exactly the same target and domain elements is already contained. To compare the basic elements, both `Dom_T` and `Tar_T` have to provide comparing functions called `equals()`. These are semantically identical such that the second function does not add new functionality to a TCB; the only difference lies in their input parameter types. To evaluate the comparison results, the ADT `Bool_T` is required, providing the constructor operators `false()` and `true()`, as well as a logic conjunction operator called `and()`. Additionally, `Elem_T` provides the operator `getFirst()`, returning the domain element, and `getSecond()`, returning the target element to extract individual tuple elements from a pair. Finally, an element can be inserted into the container by `insert()` as already known from *Set*. All operators are encapsulated by an additional interface function `r_insert()` of `Rel_T` except for `create()`, which is required to construct the second input parameter of the interface function.

2. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$
 $\wedge\ \exists s_{i,A} : param\_sort(s_{i,A}, A_R)$
 $\wedge\ \exists! s_{j,A} : relation\_sort(s_{j,A})$
 $\wedge\ i \neq j$
 $\wedge\ relation\_sort(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	 `Elem_T create(Dom_T d, Tar_T t),`
	 `Cont_T insert(Cont_T c, Elem_T e),`
	 `Bool_T equals(Dom_T d`$_1$`, Dom_T d`$_2$`),`
	 `Dom_T getFirst(Elem_T e),`
	 `Tar_T getSecond(Elem_T e),`
	 `Bool_T false(),`
	 `Bool_T true(),`
	 `Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	 $\beta(\{next, insert, equals, getFirst, getSecond, false, true, and\}) := \{\cup\}$
	 $\beta(\{r\_insert\}) := \{\cup\}$ (Interface Function)

The next rule aims at an algebra operation (called $r\_\in()$ in Table 4.3) that checks whether a pair of domain and target elements is contained by a given relation. Though this operation semantically equals the universe operator $\in$ (just like $s\_\in()$), one input parameter now is of sort *Relation*, and two other input parameters belong to the parameter sorts of A_R. This difference results in a different TCB function set compared to the one derived from $s\_\in()$. While it also contains the ADT operators `next()`, `false()`, and `true()`, it requires additional operators that deal with pairs (2. rule). A new interface function `r_isElement()` is added to `Rel_T` called, encapsulating these functions. Moreover, the operator `create()` is necessary to construct the input parameters of `r_isElement()`.

3. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \in$
 $\wedge\ \exists! s_{i,A} : \mathit{relation\_sort}(s_{i,A})$
 $\wedge\ \exists s_{j,A} : \mathit{param\_sort}(s_{j,A}, A_R)$
 $\wedge\ i \neq j$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	 `Dom_T create(),`
	 `Bool_T equals(Dom_T d`$_1$`, Dom_T d`$_2$`),`
	 `Domain_T getFirst(Elem_T e),`
	 `Target_T getSecond(Elem_T e),`
	 `Bool_T false(),`
	 `Bool_T true(),`
	 `Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	 $\beta(\{\mathit{next}, \mathit{equals}, \mathit{getFirst}, \mathit{getSecond}, \mathit{false}, \mathit{true}, \mathit{and}\}) := \{\in\}$
	 $\beta(\{\mathit{r\_isElement}\}) := \{\in\}$ (Interface Function)

The next two rules are analogous to the 4. and 5. rule for the algebra *Set*; they only differ in that `Elem_T` is a pair of basic elements, which has already been covered. Objects of type `Rel_T`, which are the input parameters for the interface function `r_diff`() and `r_union`(), have to be constructed by the constructor operators `createRelation`() and `r_insert`() of `Rel_T`. These functions have already been derived from the 1. and 2. rule, so that these rules can be reused. Note that the mapping β is equally defined as for the 2. rule. For this reason, it is not contained in the 4. rule again.

4./5. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$ $\wedge\ \forall s_{i,A} : \mathit{relation\_sort}(s_{i,A})$	$\varphi(o_A) := \{$ `Rel_T createRelation(),` `Rel_T r_insert(Rel_T r, Elem_T e)`$\}$ $\beta(\{\mathit{r\_union}\}) := \{\cup\}$ (Interface Function)

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \setminus$
 $\wedge\ \forall s_{i,A} : \mathit{relation\_sort}(s_{i,A})$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	 `Rel_T createRelation(),`
	 `Bool_T equals(Dom_T d`$_1$`, Dom_T d`$_2$`),`
	 `Rel_T r_insert(Cont_T r, Elem_T e),`
	 `Cont_T delete(Cont_T c, Elem_T e),`
	 `Dom_T getFirst(Elem_T e),`
	 `Tar_T getSecond(Elem_T e),`
	 `Bool_T false(),`
	 `Bool_T true(),`
	 `Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	 $\beta(\{\mathit{next}, \mathit{equals}, \mathit{delete}, \mathit{getFirst}, \mathit{getSecond}, \mathit{false}, \mathit{true}, \mathit{and}\}) := \{\setminus\}$
	 $\beta(\{\mathit{r\_diff}\}) := \{\setminus\}$ (Interface Function)

The remaining rules are concerned with *Relation*-specific operations that modify a relation's domain and codomain. Both rules have in common that they can be applied to two algebra operations at a time, since these operations require the same TCB functions. More precisely, rule six can be applied to operations that either subtract or restrict the domain of a relation. These operations can be distinguished from similar ones of *Mapping* by exactly one input parameter that must be of sort *Relation* and an output parameter of sort *Relation*. Both operations then require eight TCB functions; the semantical difference of the operations becomes apparent in a TCB's implementation by using sequence control mechanisms in differing ways. Result is a set of ADT operators, out of which six functions are encapsulated by two interface functions `r_domainSub()` and `r_domainRes()` provided by `Rel_T`. Both interface functions require an input parameter of type `Set_T`. The functions `createSet()` and `s_insert()` are necessary to create the input parameter of type `Set_T` and insert elements; they require additional ADT operators, e.g., `insert`() defined on `Con_T`, which have already been discussed by the 1.and 2. rule of *Set*.

6. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = ⩤$	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
$\vee\, \alpha(o_A) = \lhd)$	`Set_T createSet(),`
$\wedge\, \exists!\, s_{i,A} : relation\_sort(s_{i,A})$	`Set_T s_insert(Set_T s,Elem_T e),`
$\wedge\, relation\_sort(s_A)$	`Bool_T equals(Dom_T` d_1`,Dom_T` d_2`),`
	`Cont_T delete(Cont_T c, Elem_T e),`
	`Dom_T getFirst(Elem_T e),`
	`Bool_T false(),`
	`Bool_T true()`$\}$
	$\beta(\{next, equals, delete, getFirst, false, true\}) := \{⩤, \lhd\}$
	$\beta(\{r\_domainSub\}) := \{⩤\}$ (Interface Function)
	$\beta(\{r\_domainRes\}) := \{\lhd\}$ (Interface Function)

While the 6. rule is defined on domain subtraction and restriction, the following rule is concerned with subtracting and restricting a relation's codomain. Besides the semantical difference, which is covered by the mapping α, a small difference lies in the derived TCB function set: the manipulation of a relation's range requires a function to select the target elements of pairs (`getSecond()`) rather than to select the domain elements (`getFirst()`). This implies that the comparing operator `equals()` is now defined on `Tar_T` instead of `Dom_T`. As already mentioned, the latter does not impose a semantically different TCB function. `Rel_T` needs to provide two interface functions, called `r_domainSub()` and `r_domainRes()`.

7. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = ⩥$	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
$\vee\, \alpha(o_A) = \rhd)$	`Set_T createSet(),`
$\wedge\, \exists!\, s_{i,A} : relation\_sort(s_{i,A})$	`Set_T s_insert(Set_T s,Elem_T e),`
$\wedge\, relation\_sort(s_A)$	`Bool_T equals(Dom_T` d_1`,Dom_T` d_2`),`
	`Cont_T delete(Cont_T c, Elem_T e),`
	`Tar_T getSecond(Elem_T e),`
	`Bool_T false(),`
	`Bool_T true()`$\}$
	$\beta(\{next, equals, delete, getSecond, false, true\}) := \{⩥, \rhd\}$
	$\beta(\{r\_rangeSub\}) := \{⩥\}$ (Interface Function)
	$\beta(\{r\_rangeRes\}) := \{\rhd\}$ (Interface Function)

In case a model-specific algebra *Relation* defines all operations, for which we have defined formal rules, the result are eight ADTs as described below. `Relation[Domain,Target]` is a parameterized ADT defining the sort `Rel_T` with nine interface operators (Table 4.14). It imports the sorts `Bool_T`, `Dom_T`, `Tar_T`, and `Set_T` to define the input and output parameters of its operators. In analogy to `Set_T`, the semantics of the operators is specified by universe operations defined by the mapping α.

sorts	`Rel_T`
import	`Bool_T`, `Dom_T`, `Tar_T`, `Set_T`

operators	`createRelation:` $\rightarrow$ `Rel_T`
	`r_insert:` `Rel_T` $\times$ `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Rel_T`
	`r_isElement:` `Rel_T` $\times$ `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Bool_T`
	`r_union:` `Rel_T` $\times$ `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Rel_T`
	`r_diff:` `Rel_T` $\times$ `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Rel_T`
	`r_domainSub:` `Rel_T` $\times$ `Set_T` $\rightarrow$ `Rel_T`
	`r_rangeSub:` `Rel_T` $\times$ `Set_T` $\rightarrow$ `Rel_T`
	`r_domainRes:` `Rel_T` $\times$ `Set_T` $\rightarrow$ `Rel_T`
	`r_rangeRes:` `Rel_T` $\times$ `Set_T` $\rightarrow$ `Rel_T`

Table 4.14: ADT `Relation[Domain,Target]` (`Rel_T`)

The ADTs `Dom_T` and `Tar_T` define the basic tuple element types of a relation, just like `Elem_T` has defined the basic element type of `Set_T`. For each the rule system derives a constructing function (`create()`) as well as a comparing function (`equals()`). As already mentioned, `Dom_T` and `Tar_T` are identical to `Elem_T` of `Set_T` except for a different name (Table 4.11).

The new ADT `Elem_T` (Table 4.15) is now a representative for tuple elements. As such it has tuple-specific selectors (`getFirst()`, `getSecond()`) to individually access both tuple elements. The constructor semantically remains the same even though it now has two input parameters.

sorts	`Elem_T`
import	`Dom_T`, `Tar_T`

operators	`create:` `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Elem_T`
	`getFirst:` `Elem_T` $\rightarrow$ `Dom_T`
	`getSecond:` `Elem_T` $\rightarrow$ `Tar_T`

variables	$\forall \mathtt{d} : \mathtt{Dom\_T}, \mathtt{t} : \mathtt{Tar\_T}$

axioms	`getFirst(create(d,t)) = d`
	`getSecond(create(d,t)) = t`

Table 4.15: Modified ADT Element (`Elem_T`)

Compared to the `Bool_T` ADT derived from algebra *Set* (Table 4.10), the ADT `Bool_T` derived from *Relation* is enhanced by one operator `and()` and additional axioms for the operator's interpretation. The enhanced ADT is presented in Table 4.16.

sorts	`Bool_T`

operators	`true:` $\rightarrow$ `Bool_T`
	`false:` $\rightarrow$ `Bool_T`
	`and:` `Bool_T` $\times$ `Bool_T` $\rightarrow$ `Bool_T`

variables	$\forall b_1, b_2$: `Bool_T`

axioms	`true() = true`
	`false() = false`
	`and(`b_1`,false) = false`
	`and(`b_1`,true) = b`

Table 4.16: Modified ADT `Bool_T`

The ADT `Set_T` (Table 4.17) is required to construct input parameters of type `Set_T` for the interface operators `r_domainSub()`, `r_domainSub()`, `r_rangeSub()`, `r_domainRes()`, and `r_rangeRes()` (6./7. rule). As such, the ADT only provides two interface functions, whose semantics are defined by the mapping β. Since it imports `Elem_T` to define the input parameters of the operator `s_insert()`, we also have to specify the ADT `Elem_T` as basic element type for `Set_T`. As the rule system does not define additional operators for `Elem_T` than compared to the rule system for *Set*, we can refer to the one presented in Table 4.11. The same holds for the ADT `Cont_T`, which equals the corresponding one derived from *Set*; it is illustrated by Table 4.12, but lacking the function `delete()`.

sorts	`Set_T`
import	`Elem_T`

operators	`createSet:` $\rightarrow$ `Set_T`
	`s_insert:` `Set_T` $\times$ `Elem_T` $\rightarrow$ `Set_T`

Table 4.17: Modified ADT `Set[Element]` (`Set_T`)

Here, we now have two conflicts: on the one hand, `Elem_T` represents tuple elements for `Rel_T`, on the other hand it is the basic type for `Set_T`. The same holds for `Cont_T`; being a container for tuple elements and for basic elements. These conflicts are solved by using inheritance and polymorphism. To this end, `Elem_T` becomes an abstract ADT (Figure 4.22), to define the required function interfaces and from which tuple elements `TElem_T` are derived. Both derived element types share two TCB functions, the constructor and a comparison function, which only differ in their implementations. All other functions are element-specific and contained by the individual element types, e.g., `getFirst()` and `getSecond()`. By this means, we avoid functional redundancy of different container and element types.

The interrelations of all ADTs derived from the algebra *Relation* are shown in a nutshell in Figure 4.22. As can be seen, both `Set_T` and `Rel_T` existentially depend on `Cont_T`, which is associated with the abstract `Elem_T` by an arbitrary number of objects, from which tuple elements `TElem_T` can be derived. The namespace `SET` indicates that `Cont_T` and `Elem_T` are reused. Objects of `TElem_T` are constructed by objects of the basic types `Dom_T` and `Tar_T`. As already mentioned, for the sake of type safety we distinguish between two basic types here; in an implementation they may collapse into a single one since they all provide the same operators. All basic types use the operators of ADT `Bool_T` to implement their operators.

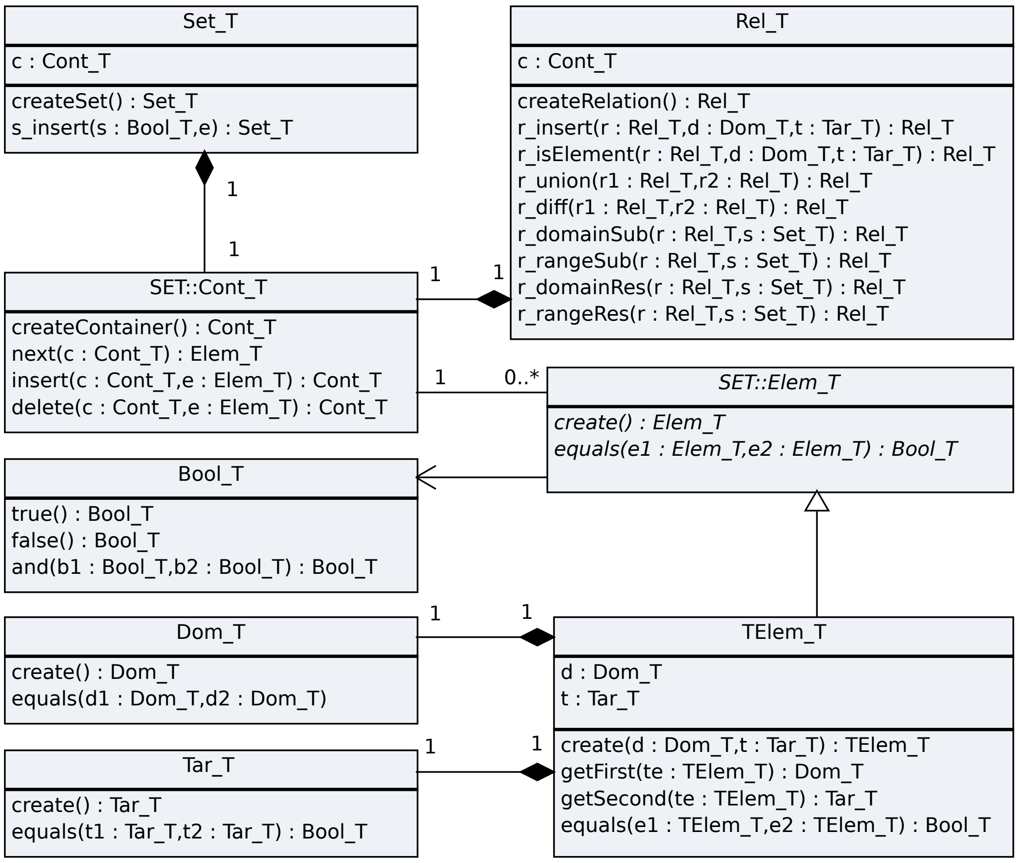
Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Bool_T,e) : Set_T
Rel_T
c : Cont_T
createRelation() : Rel_T
r_insert(r : Rel_T,d : Dom_T,t : Tar_T) : Rel_T
r_isElement(r : Rel_T,d : Dom_T,t : Tar_T) : Rel_T
r_union(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_diff(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_domainSub(r : Rel_T,s : Set_T) : Rel_T
r_rangeSub(r : Rel_T,s : Set_T) : Rel_T
r_domainRes(r : Rel_T,s : Set_T) : Rel_T
r_rangeRes(r : Rel_T,s : Set_T) : Rel_T
1
1
SET::Cont_T
createContainer() : Cont_T
next(c : Cont_T) : Elem_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
1
1
1
0..*
SET::Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T
Bool_T
true() : Bool_T
false() : Bool_T
and(b1 : Bool_T,b2 : Bool_T) : Bool_T
1
1
Dom_T
create() : Dom_T
equals(d1 : Dom_T,d2 : Dom_T)
TElem_T
d : Dom_T
t : Tar_T
create(d : Dom_T,t : Tar_T) : TElem_T
getFirst(te : TElem_T) : Dom_T
getSecond(te : TElem_T) : Tar_T
equals(e1 : TElem_T,e2 : TElem_T) : Bool_T
1
1
Tar_T
create() : Tar_T
equals(t1 : Tar_T,t2 : Tar_T) : Bool_T

Figure 4.22: ADTs Derived from $Relation[Domain, Target]$

Result is a formal rule system that describes causal dependencies between security models in algebra notation and TCB functions represented by ADTs. The rule system provides a set of predefined TCB functions, from which each rules selects the functions that are composed by a specific policy-dependent RTE. The derived set of TCB functions consists of two ADT types: (i) ADTs that are directly produced by the component algebras such as `Set_T` and `Rel_T` and that provide interface functions, and (ii) ADTs that are derived from the algebras' operations, providing basic operators to implement the interface functions. If these ADT operators are not sufficient for implementing for instance novel security models, the algebra foundation needs to be extended by new operations or even new algebras, which involves extending the rule system, too.

The following section applies the formal rule system to the RBAC HIS policy and shows that even for this real-world based example the algebra foundation along with the formal rule system is sufficient.

4.4.5 Summary

This section has presented an engineering approach for the policy-dependent RTE of a TCB. Based on rewriting a core-based model in algebraic notation, TCB functions are derived by exploiting causal dependencies between security models and TCB functions. Causal dependencies are defined by a formal rule system whose input is a set of algebras that represent a core-based model. Output of the rule system is a set of ADTs, describing the functions of the `security model functions` component. By applying the concepts of subsuming security models, inheritance, and polymorphism, the derived function set is nonredundant as far as the trade-off with type safety allows. The functional range of a TCB's `interceptor` is defined by a security policy's authorization scheme.

4.5 Policy-dependent RTE for the RBAC HIS Policy

After having presented the engineering method for a TCB's policy-dependent RTE along with causal dependencies between security models and TCB functions in an abstract way in Section 4.4, the goal of this section is to demonstrate both by means of the real world-based RBAC HIS policy introduced in Section 3.4.4. Moreover, this section aims at showing that the approach of engineering causal TCBs is feasible. For this purpose, this section engineers a causal TCB that supports the RBAC HIS policy by specifying the functionality of the components `security model functions` and `interceptor` of the policy-dependent RTE and discussing the interactions of the `executable policy` with these TCB components. Figure 4.23 shows the design of a policy-dependent RTE that is tailored to enforce the RBAC HIS policy. Due to demonstration purposes, the TCB will be engineered to support only this policy. Hence, the policy-dependent RTE contains only a single component `security model functions` that implements this RBAC model, called `RBAC security model functions`, and an `interceptor` component that is matched by the `executable RBAC HIS policy`.

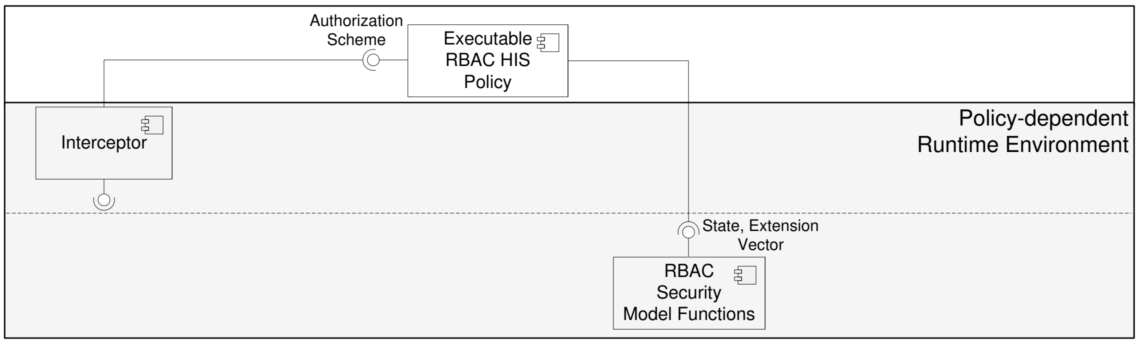
Authorization Scheme
Executable RBAC HIS Policy
Interceptor
Policy-dependent Runtime Environment
State, Extension Vector
RBAC Security Model Functions

Figure 4.23: Policy-dependent RTE Tailored to Enforce the RBAC HIS Policy

The section is structured as follows. Section 4.5.1 addresses the functions of the `RBAC security model functions` component and shows how they are derived from the underlying RBAC model. Afterwards, Section 4.5.2 discusses the proxy functions of the interceptor for the `executable RBAC policy`. The section closes with presenting the `executable RBAC HIS policy` and showing its interactions with the components of the policy-dependent RTE (Section 4.5.3).

4.5.1 RBAC Security Model Functions

The goal of this section is to demonstrate the derivation of TCB functions from a formal security model by means of the RBAC HIS policy example. For this purpose, we stepwise apply the TCB engineering method presented in Section 4.4.1. Input of the TCB engineering method is the core-based RBAC security model described in Section 3.4.5; output is a set of TCB functions that implement this model and is described by means of ADTs.

4.5.1.1 Algebra Engineering

The first step of engineering a policy-dependent RTE is to rewrite the security model in algebraic notation. Here, we first have to define the component algebras of the RBAC model; the core algebras *State* and *ExtensionVector* are specified afterwards. In order to define the component algebras, we need to consider the set of model components $M = \{U, S, R, UA, user, roles, O, OP, RH, RE, m\}$. While U, S, R, O, and OP are of mathematical type set, UA, RH, and RE are relations, and $user$, $roles$, and m are unary and binary mappings. As can be seen, even for this real-world example, the four basic component algebras of the universal algebra foundation are sufficient. Hence, there is no need to define additional algebras here; it remains to define the algebra's base sets and operation sets with respect to the RBAC model.

To begin with, we specify the component algebra $Set[Element]$ as algebraic representation of the model components U, S, R, O, and OP (Table 4.18). The base set of the parameter sort *Element* is defined as a union of these model sets so that the base set of *Set* then specifies a finite subset of the base set of *Element*.

The operations of *Set* depend on the semantics of the model's primitive actions and clauses. Let us first consider primitive actions. The user set U and the session set S are dynamic model components that are part of the model state. By means of four primitive actions (*addUsers/deleteUsers* and *createSessions/destroySessions*), new users and sessions can be added and already existing users and sessions can be deleted. In terms of set theory, these primitive actions apply the union and the difference operator of sets (see Appendix A.3). Thus, the operation set must contain two operations ($s\_\cup()$ and $s\_\backslash()$), implementing these set operators.

The remaining sets R, O, and OP are static so that there are no primitive actions to modify them. However, the model provides clauses that check whether a given element is in a specific set, e.g., $clause_m$ returns true if a given operation is element of a subset of the operation set OP. For this reason, the algebra also requires an operation called $s\_\in()$ that implements the set operator $\in$. In addition, *Set* needs constructors which allow for creating all objects of the sort *Set*. This responsibility is met by the operations $s\_\varnothing()$ and $s\_insert()$. With respect to the RBAC model, the operation set of *Set* is now complete.

import	$Bool \cong \{true, false\}$ $Element \cong e \in E := U \cup S \cup R \cup O \cup OP$, nonempty set of elements
sorts	Set $\cong S \subseteq E, S$ is finite
operations	$s\_\varnothing :\to Set$ $s\_insert : Set \times Element \to Set$ $s\_\in: Set \times Element \to Bool$ $s\_\cup : Set \times Set \to Set$ $s\_\backslash : Set \times Set \to Set$
semantics	$s\_\varnothing() := \varnothing$ $s\_insert(S, e) := S \cup \{e\}$ $s\_\in(S, e) := \begin{cases} true, & e \in S \\ false, & e \notin S \end{cases}$ $s\_\cup(S, S') := S \cup S'$ $s\_\backslash(S, S') := S \backslash S'$

Table 4.18: Component Algebra $Set[Element]$ for the RBAC Model

It remains to specify the sort *Element* as parameter sort (Table 4.19). The only operations of the algebra *Element* are the constructor *create*() and an operation *equals*() to verify the equality of two elements. The latter is required by container sorts such as *Set*, in order to implement operations like $s\_insert()$ or $s\_ \in ()$.

import	$Bool \cong \{true, false\}$
sorts	$Element \cong e$ element of a nonempty set (Universe)
operations	$create :\to Element$ $equals : Element \times Element \to Bool$
semantics	$create() = e$ $equals(e_1, e_2) := \begin{cases} true, & e_1 = e_2 \\ false, & e_1 \neq e_2 \end{cases}$

Table 4.19: Component Algebra for Parameter Sort *Element* for the RBAC Model

In analogy to the specification of *Set*, we now specify the algebra *Relation* (Table 4.20) as algebraic representation of the relations UA, RH, and RE. The latter are binary relations so that two parameter sorts *Domain* and *Target* are sufficient. Since these are equivalent to the parameter sort *Element*, there is no need to specify an additional parameter sort. As mentioned in Section 4.4.2, the base sets of *Domain* and *Target* depend on the relations' domains and codomains of the model. More precisely, while the base set of *Domain* is defined as union of all domains of UA_q, RH, and RE, the base set of *Target* specifies a union of all codomains of these relations.

import	$Bool \cong \{true, false\}$ $Domain \cong D := U \cup R = \{d_1, \ldots, d_i, \ldots, d_n\}, D \neq \varnothing, n \in \mathbb{N}, 1 \leqslant i \leqslant n$ $Target \cong T := R = \{t_1, \ldots, t_j, \ldots, t_m\}, T \neq \varnothing, m \in \mathbb{N}, 1 \leqslant j \leqslant m$ $Set[Element] \cong S \subseteq D \cup T, S$ is finite
sorts	$Relation \cong Rel \subseteq D \times T, Rel$ is finite
operations	$r\_\varnothing :\to Relation$ $r\_insert : Relation \times Domain \times Target \to Relation$ $r\_\in: Relation \times Domain \times Target \to Bool$ $r\_\cup : Relation \times Relation \to Relation$ $r\_\backslash : Relation \times Relation \to Relation$ $r\_\lhd : Relation \times Set[Element] \to Relation$
semantics	$r\_\varnothing() := \varnothing$ $r\_insert(Rel, d_i, t_j) := Rel \cup \{(d_i, t_j)\}$ $r\_\in (Rel, d_i, t_j) := \begin{cases} true, & (d_i, t_j) \in Rel \\ false, & (d_i, t_j) \notin Rel \end{cases}$ $r\_\cup (Rel, Rel') := Rel \cup Rel'$ $r\_\backslash(Rel, Rel') := Rel \backslash Rel'$ $r\_\lhd (Rel, \{d_1, \ldots, d_i\}) := \{(d, t) \mid (d, t) \in Rel \wedge d \notin \{d_1, \ldots, d_i\})\}$

Table 4.20: Component Algebra *Relation*[*Domain*, *Target*] for the RBAC Model

Just as the operation set of any other component algebra, the operation set of *Relation* also depends on the model's primitive actions and clauses. The state component *UA* can be modified by three primitive actions (*deleteUsers, assignUserToRoles*, and *revokeUserFromRoles*), which apply the standard set operators $\backslash$ and $\cup$. Moreover, *deleteUsers* uses a relation-specific operator $\lhd$, called domain subtraction. For these reasons, the algebra defines the operations $r\_\backslash()$, $r\_\cup()$, and $r\_\lhd()$ as projectors on the sort *Relation* that exactly implement the primitives' operators. This results in that the already defined sort *Set* has to be imported, since one of the import parameters of $r\_\lhd()$ must be of sort *Set*. Its base set is the subset of the union of the base sets of *Domain* and *Target*. Besides these operations, *Relation* also provides the operation $r\_\in()$ to check whether a pair of domain element and target element is contained in a relation. This operation is required for supporting the clauses $clause_{UA}$, $clause_{RH}$, and $clause_{RE}$. Moreover, the algebra needs two constructors ($r\_\varnothing()$ and $r\_insert()$) to be able to create any object of the sort *Relation*.

The next step is to specify algebraic representations for the model components *user*, *roles*, and *m*. *user* and *roles* are unary mappings; they can thus be represented by the algebra *Mapping*. On the contrary, *m* is a binary mapping modeling an ACM so that it must be rewritten by the algebra *Matrix*. In the following, we provide details on the component algebra *Mapping* for the RBAC model (Table 4.21); details on the algebra *Matrix* will be given afterwards.

Analogous to the specification of *Relation*, we define the base sets of the sorts *Domain* and *Target* of the algebra *Mapping* by unifying the domains and codomains of the mappings *user* and *roles*. Note that the base set of *Target* is a union of U and 2^R as defined by the codomains of *user* and *roles*. Again, the sort *Set* needs to be imported since it is required as input parameter sort of two commands; its base set combines the base sets of *Domain* and *Target*.

import	$Domain \cong D := S = \{d_1, \ldots, d_i, \ldots, d_n\}, D \neq \varnothing, n \in \mathbb{N}, 1 \leqslant i \leqslant n$ $Target \cong T := U \cup \mathscr{P}(R) = \{t_1, \ldots, t_j, \ldots, t_m\}, T \neq \varnothing, m \in \mathbb{N}, 1 \leqslant j \leqslant m$ $Set[Element] \cong S \subseteq D \cup T, S$ is finite
sorts	$Mapping \cong Map \subseteq D \times T, Map$ is finite
operations	$mp\_\varnothing :\rightarrow Mapping$ $mp\_init : Mapping \rightarrow Mapping$ $mp\_insert : Mapping \times Domain \times Target \rightarrow Mapping$ $mp\_getMapping : Mapping \times Domain \rightarrow Set[Target]$ $mp\_\backslash : Mapping \times Mapping \rightarrow Mapping$ $mp\_\lhd : Mapping \times Set[Element] \rightarrow Mapping$ $mp\_\rhd : Mapping \times Set[Element] \rightarrow Mapping$ $mp\_\oplus : Mapping \times Mapping \rightarrow Mapping$
semantics	$mp\_\varnothing() := \varnothing$ $mp\_init(Map) := \{(d,t) \mid (d,t) \in Map\}$ $mp\_insert(Map, d_i, t_j) := \begin{cases} Map \cup \{(d_i, t_j)\}, & \nexists (d,t) \in Map : eq(d_i, d) \\ Map, & otherwise \end{cases}$ $mp\_getMapping(Map, d_i) := \begin{cases} \{t_j\}, & (d_i, t_j) \in Map \\ s\_\varnothing(), & (d_i, t_j) \notin Map \end{cases}$ $mp\_\backslash(Map, Map') := Map \backslash Map'$ $mp\_\lhd(Map, \{d_1, \ldots, d_i\}) := \{(d,t) \mid (d,t) \in Map \wedge d \notin \{d_1, \ldots, d_i\})\}$ $mp\_\rhd(Map, \{t_1, \ldots, t_j\}) := \{(d,t) \mid (d,t) \in Map \wedge t \notin \{t_1, \ldots, t_j\})\}$ $mp\_\oplus(Map, Map') := \{(d,t) \mid (d,t) \in Map \wedge d \notin D')\} \cup Map'$, where D' is the domain of Map'

Table 4.21: Component Algebra *Mapping*[*Domain*, *Target*] for the RBAC Model

Since *user* and *roles* are both part of the model state, the algebra's operation set is defined by a set of primitives (*deleteUsers, createSessions, destroySessions, mapUser-Sessions, unmapUserSessions, activateRoles*, and *deactivateRoles*). Additionally, the model provides $clause_{roles}$, which selects a set of assigned roles for a given session in order to check whether this set contains a specific role. Within $clause_{roles}$, the operation performed on the mapping *roles* is to select this set of roles, while the set operator $\in$ is checking the content. Thus, only the selection operations belong to the algebra *Mapping*, while the operator $\in$ must be covered by the algebra *Set*. Result is an operation set consisting of seven operations, including constructors ($mp\_\varnothing()$, $mp\_init()$, and $mp\_insert()$), a selector ($mp\_getMapping()$), and projectors ($mp\_\backslash(), mp\_\lhd(), mp\_\rhd()$, and $mp\_\oplus()$). $mp\_init()$ is a special constructor that copies an existing mapping object to an empty one. *Set* and *Relation* have equivalent operations in the form of $s\_\cup()$ and $r\_\cup()$. $mp\_getMapping()$ returns an empty set created by the constructor of the sort *Set*, if there is no valid selection found.

The last component algebra that is required to rewrite the RBAC model in algebraic notation is *Matrix*, shown in Table 4.22. Since this specification is analogous to the above described algebra specifications, we will only point out two aspects and refrain from further explanations. (i) As already described for *Mapping*, we define the base set of *Target* by the set *OP* because the matrix definition already contains the power set of *Target*. (ii) Since m is not part of the model state, the algebra's operation set only depends on the clauses of the model. More precisely, the model contains $clause_m$ that checks whether a given operation is contained in a specific matrix cell. As with $clause_{roles}$, $clause_m$ applies two operations: selecting the content of a specific cell and checking whether this content contains a given element. The sort *Matrix* must only provide an operation to select the cell content; the checking is the responsibility of the sort *Set*. For this reason, the operation set provides the function $m\_cellContent()$, which returns an empty set if the cell is empty. Furthermore, the algebra contains three constructors $m\_\varnothing()$, $m\_init()$ and $m\_insert()$ to create any object of the sort *Matrix*.

import	$Row \cong Ro := R = \{r_1, \ldots, d_i, \ldots, d_n\}, Ro \neq \varnothing, n \in \mathbb{N}, 1 \leqslant i \leqslant n$
	$Column \cong C := O = \{c_1, \ldots, c_i, \ldots, c_m\}, C \neq \varnothing, m \in \mathbb{N}, 1 \leqslant i \leqslant m$
	$Target \cong T := OP = \{t_1, \ldots, t_i, \ldots, t_k\}, T \neq \varnothing, k \in \mathbb{N}, 1 \leqslant i \leqslant k$
	$Set[Element] \cong S_T \subseteq T$, S is finite
sorts	$Matrix \cong M \subseteq R \times C \times 2^T, M$ is finite
operations	$m\_\varnothing :\rightarrow Matrix$
	$m\_init : Matrix \rightarrow Matrix$
	$m\_insert : Matrix \times Row \times Column \times Target \rightarrow Matrix$
	$m\_CellContent : Matrix \times Row \times Column \rightarrow Set[Target]$
semantics	$m\_\varnothing() := \varnothing$
	$m\_init(M) := \{(r, c, t) \mid (r, c, t) \in M\}$
	$m\_insert(M, r, c, t) := \begin{cases} (r', c', T \cup \{t\}), & (r', c', T) \in M : r = r' \wedge c = c' \\ M \cup \{(r, c, t)\}, & \nexists(r', c', t') \in M : r = r' \wedge c = c' \end{cases}$
	$m\_cellContent(M, r, c) := \begin{cases} S_T, & \exists!(r', c', T) \in M : r = r' \wedge c = c' \\ s\_\varnothing(), & otherwise \end{cases}$

Table 4.22: Component Algebra $Matrix[Row, Column, Target]$ for the RBAC Model

The last step of algebra engineering is to rewrite the model state and the extension vector as core algebras *State* and *ExtensionVector*; we begin with rewriting the model state. Table 4.23 shows the resulting *State* specification where the input parameter vector of the operations has already been replaced by their individual input parameter sorts and the operations are already interpreted by the operations of the component algebras for the RBAC model. As mentioned in Section 4.4.2, the sort *State* defines a tuple consisting of the base sets that represent the individual state components of a security model. In the context of the RBAC model, the model state $q = (U_q, S_q, UA_a, user_q, roles_q)$ is a tuple where U_q is the user set, S_q is the session set, UA_q is the user-to-role-mapping relation, $user_q$ is a user-to-session association mapping, and $roles_q$ is a session-roles-activation mapping (Appendix A.3). Accordingly, all component algebras that represent these five state components are imported as sorts in the algebra *State*; their base sets are then defined by the according model components. For example, the algebra $Set[User]$ is the representation of U_q where the parameter sort *User* is equivalent to the previously defined parameter sort *Element*[7]. From this it follows, that the base set of $Set[User]$ is defined by U here.

import $Bool \cong \{true, false\}$
$User \cong u := u \in U$, element of the nonempty set of users U
$Session \cong s := s \in S$, element of the nonempty set of sessions S
$Role \cong r := r \in R$, element of the nonempty set of roles R
$Set[User] \cong S_U := U = \{u_1, \ldots, u_n \mid n \in \mathbb{N}\}$
$Set[Session] \cong S_S := S = \{s_1, \ldots, s_l \mid l \in \mathbb{N}\}$
$Relation[User, Role] \cong R_{UA} \subseteq \{U \times R \mid R = \{r_1, \ldots, r_k\}, |R_{UA}| \in \mathbb{N}\}$
$Mapping[Session, User] \cong Map_{user} \subset \{S \times U \mid |Map_{user}| \in \mathbb{N}\}$
$Mapping[Session, Set[Role]] \cong Map_{roles} \subset \{S \times \mathscr{P}(R) \mid |Map_{roles}| \in \mathbb{N}\}$

sorts $State \cong q := (S_U, S_S, R_{UA}, Map_{user}, Map_{roles})$

operations $createState : Set[User] \times Set[Session] \times Relation[User, Role] \times$
$\quad Mapping[Session, User] \times Mapping[Session, Set[Role]] \rightarrow State$
$getUserS : State \rightarrow Set[User]$
$getSessionS : State \rightarrow Set[Session]$
$getUAR : State \rightarrow Relation[User, Role]$
$getRolesMap : State \rightarrow Mapping[Session, User]$
$getUserMap : State \rightarrow Mapping[Session, Set[Role]]$
$addUsers : State \times Set[User] \rightarrow State$
$deleteUsers : State \times Set[User] \rightarrow State$
$createSessions : State \times Set[Session] \rightarrow State$
$destroySessions : State \times Set[Session] \rightarrow State$
$mapUserSessions : State \times Set[Session] \times User \rightarrow State$
$unmapUserSessions : State \times Set[Session] \times User \rightarrow State$
$assignUserToRoles : State \times User \times Set[Roles] \rightarrow State$
$revokeUserFromRoles : State \times User \times Set[Roles] \rightarrow State$
$activateRoles : State \times Session \times Set[Roles] \rightarrow State$
$deactivateRoles : State \times Session \times Set[Roles] \rightarrow State$
$clause_{roles} : State \times Session \times Role \rightarrow Bool$
$clause_{UA} : State \times Role \times User \rightarrow Bool$

semantics $createState(S_U, S_S, R_{UA}, Map_{user}, Map_{roles}) := (s\_ \cup (\varnothing, S_U), s\_ \cup (\varnothing, S_R),$
$\quad r\_ \cup (\varnothing, R_{US}), mp\_init(Map_{user}),$
$\quad mp\_init(Map_{roles}))$
$getUserS(q) := S_U$
$getSessionS(q) := S_S$
$getUAR(q) := R_{UA}$

[7]In order to enable a proper readability, we have renamed the parameter sorts based on the names of model components.

$$
\begin{aligned}
&getRolesMap(q) := Map_{roles}\\
&getUserMap(q) := Map_{user}\\
&addUsers(q, \{u_1, \ldots, u_m\}) := (s\_ \cup (S_U, \{u_1, \ldots, u_m\}), S_S, R_{UA}, Map_{user}, Map_{roles})\\
&deleteUsers(q, \{u_1, \ldots, u_m\}) := (s\_\backslash(S_U, \{u_1, \ldots, u_m\}), S_S,\\
&\qquad r\_ \lhd (R_{UA}, \{u_1, \ldots, u_m\}),\\
&\qquad mp\_ \rhd (Map_{user}, \{u_1, \ldots, u_m\}), Map_{roles})\\
&createSessions(q, \{s_1, \ldots, s_m\}) := (S_U, s\_ \cup (S_S, \{s_1, \ldots, s_m\}), R_{UA}, Map_{user},\\
&\qquad mp\_ \ \oplus\\
&(Map_{roles}, \{(x_{s_1}, s\_\varnothing()), ..., (s_m, s\_\varnothing())\}))\\
&destroySessions(q, \{s_1, \ldots, s_m\}) := (S_U, s\_\backslash(S, \{s_1, \ldots, s_m\}), R_{UA},\\
&\qquad mp\_ \lhd (Map_{user}, \{s_1, \ldots, s_m\}), \{s_1, \ldots, s_m\}))\\
&mapUserSessions(q, \{s_1, \ldots, s_m\}, u) := (S_U, S_S, R_{UA}, mp\_ \oplus (Map_{user},\\
&\qquad \{(s_1, u), \ldots, (s_m, u)\}), Map_{roles})\\
&unmapUserSessions(q, \{s_1, \ldots, s_m\}, u) := (S_U, S_S, R_{UA}, mp\_\backslash(Map_{user},\\
&\qquad \{(s_1, u), \ldots, (s_m, u)\}), Map_{roles})\\
&assignUserToRoles(q, u, \{r_1, \ldots, r_m\}) := (S_U, S_S, r\_ \cup (R_{UA}, \{(u, r_1), \ldots, (u, r_m)\}),\\
&\qquad Map_{user}, Map_{roles})\\
&revokeUserFromRoles(q, u, \{r_1, \ldots, r_m\}) := (S_U, S_S, r\_\backslash(R_{UA}, \{(u, r_1), \ldots,\\
&\qquad (u, r_m)\}), Map_{user}, Map_{roles})\\
&activateRoles(q, s, \{r_1, \ldots, r_m\}) := (S_U, S_S, R_{UA}, Map_{user}, mp\_ \oplus (Map_{roles}, \{(s,\\
&\qquad mp\_insert(Map_{roles}, s, \{r_1, \ldots, r_m\}))\}))\\
&deactivateRoles(q, s, \{r_1, \ldots, r_m\}) := (S_U, S_S, R_{UA}, Map_{user}, mp\_ \oplus (Map_{roles},\\
&\qquad mp\_\backslash(Map_{roles}, \{(s, \{r_1, \ldots, r_m\})\})))\\
&clause_{roles}(q, s, r) := \begin{cases} true, & s\_ \in (mp\_getMapping(Map_{roles}, s), r) \\ false, & otherwise \end{cases}\\
&clause_{UA}(q, r, u) := \begin{cases} true, & r\_ \in (R_{UA}, \{u, r\}) \\ false, & otherwise \end{cases}
\end{aligned}
$$

Table 4.23: Core Algebra *State* for the RBAC Model

As can be seen in Table 4.23, the operation set specifies a constructor, a selector operation for each tuple element, the model's set of primitive actions plus the clauses $clause_{roles}$ and $clause_{UA}$, which are defined on the state components according to their names. In the following, we focus on rewriting the operation *createSessions*, since it comprises a couple of interesting aspects, which are scattered over the other operations.

1. $createSessions : State \times Set[Session] \rightarrow State$
2. $createSession : (q, \{s_1, \ldots, s_n\}) := (S_U, S_S \cup \{s_1, \ldots, s_n\}, R_{UA}, Map_{user}, Map_{roles} \oplus \{(s_1, \varnothing), ..., (s_n, \varnothing)\})$
3. $createSessions(q, \{s_1, \ldots, s_m\}) := (S_U, s\_ \cup (S_S, \{s_1, \ldots, s_m\}), R_{UA}, Map_{user}, mp\_ \oplus (Map_{roles}, \{(x_{s_1}, s\_\varnothing()), ..., (s_m, s\_\varnothing())\}))$

createSession generates a new session and activates an empty set of roles for this session. As soon as roles are activated for this session by *activateRoles*, the empty role set is replaced by a specific role set. The input vector X is replaced by the individual input parameter S, which again is represented by the sort $Set[Session]$ (1. step). The 2. step rewrites the model components that are still part of the operation's semantics by their algebraic representations, e.g., U_q is rewritten by S_U and $user_q$ is rewritten by Map_{user}. In the last step, the operations of the above defined component algebras are applied to rewrite the interpretation of *createSessions* in algebraic notation. Instead of $S_S \cup \{s_1, \ldots, s_n\}$, we now write $s\_\cup(S_S, \{s_1, \ldots, s_m\})$, which is the union operator defined for the algebra $Set[Element]$. This step also includes that the empty set in Map_{roles} is replaced by the constructor of the sort *Set*. This approach is applied to any primitive action and state-based clause of a security model.

import	$Bool \cong \{true, false\}$
	$Operation \cong op := op \in OP$, element of the nonempty set of operations OP
	$Role \cong r := r \in R$, element of the nonempty set of roles R
	$Object \cong o := o \in O$, element of the nonempty set of objects O
	$Set[Object] \cong S_O := \{o_1, \ldots, o_n \mid n \in \mathbb{N}\}$
	$Set[Operation] \cong S_{OP} := \{op_1, \ldots, op_m \mid m \in \mathbb{N}\}$
	$Set[Role] \cong S_R := \{r_1, \ldots, r_k \mid k \in \mathbb{N}\}$
	$Matrix[Role, Object, Set[Operation]] \cong M_m \subset \{R \times O \times \mathscr{P}(OP) \| \|m\| \in \mathbb{N}\}$
	$Relation[Role, Role] \cong R_{RH} \subseteq \{R \times R \mid \|RH\| \in \mathbb{N}\}, R_{RE} \subseteq \{R \times R \mid \|RE\| \in \mathbb{N}\}$
sorts	$ExtensionVector \cong E := (S_O, S_{OP}, S_R, M_m, R_{RH}, R_{RE})$
operations	$createExtVec : Set[Object] \times Set[operation] \times Set[Role] \times$ $Matrix[Role, Object, Set[Operation]] \times Relation[Role, Role] \times$ $Relation[Role, Role] \rightarrow State$
	$getObjectS :\rightarrow Set[Object]$
	$getOperationS :\rightarrow Set[Operation]$
	$getRoleS :\rightarrow Set[Role]$
	$getMatrixM :\rightarrow Matrix[Role, Object, Set[Operation]]$
	$getRHR :\rightarrow Relation[Role, Role]$
	$getRER :\rightarrow Relation[Role, Role]$
	$clause_m : Operation \times Role \times Object \rightarrow Bool$
	$clause_{RH} : Role \times Role \rightarrow Bool$
	$clause_{RE} : Role \times Role \rightarrow Bool$
semantics	$createExtVec(S_O, S_{OP}, S_R, M_m, R_{RH}, R_{RE}) := (s\_\cup(\varnothing, S_O), s\_\cup(\varnothing, S_{OP}), s\_\cup(\varnothing, S_R), m\_init(createMatrix(), M), r\_\cup(\varnothing, R_{RH}), r\_\cup(\varnothing, R_{RE}))$
	$getObjectS() := S_O$
	$getOperationS() := S_{OP}$
	$getRoleS() := S_R$
	$getMatrixM() := M_m$
	$getRHR() := R_{RH}$
	$getRER() := R_{RE}$
	$clause_m(op, r, o) := \begin{cases} true, & s\_\in(m\_cellContent(M_m, r, o), op) \\ false, & otherwise \end{cases}$
	$clause_{RH}(r_1, r_2) := \begin{cases} true, & r\_\in(R_{RH}, r_1, r_2) \\ false, & otherwise \end{cases}$
	$clause_{RE}(r_1, r_2) := \begin{cases} true, & r\_\in(R_{RE}, r_1, r_2) \\ false, & otherwise \end{cases}$

Table 4.24: Core Algebra for *ExtensionVector* for the RBAC Model

The procedure of rewriting the model's extension vector as algebra *ExtensionVector* is similar. The resulting specification is illustrated in Table 4.24. The only exception is the specification of the algebra's operation set, which consists besides the constructor and selector operations only of those clauses that are defined on elements of the extension vector. Here, these clauses are $clause_m$, $clause_{RH}$, and $clause_{RE}$. $clause_m$ is special in that it has to check whether a given operation is contained by a specific matrix cell. For this purpose, the operation *m_cellContent*() of the sort *Matrix* is used to extract the contents of the given matrix cell. Afterwards, the operation $s\_ \in ()$ of the sort *Set* checks if the given operation is contained by the returned set.

The result of algebra engineering is an algebraic representation of the RBAC model, containing four component algebras *Set*, *Relation*, *Mapping*, and *Matrix*, and the two core algebra *State* and *ExtensionVector*. This provides the basis for deriving the TCB functions of the policy-depend RTE by ADT engineering.

4.5.1.2 ADT Engineering

ADT engineering follows three steps; we now apply each step to the core-based model that formalized the RBAC HIS policy (Appendix B):

1. Designing model-specific basic types,
2. Applying the formal rule system to the model-specific component algebras, and
3. Rewriting model-specific core algebras in corresponding ADTs.

The formal rule system described in Section 4.4.4 uses model-independent types such as `Elem_T`, `Dom_T`, or `Tar_T`, as basic types. However, this is not sufficient for deriving TCB functions for a specific security model, since all model-specific basic types would be subsumed by these model-independent types, which results in undermining the type safety of the model implementation. For this reason, we design model-specific basic types based on the basic elements of a model. In doing so, we set the course for a type-safe implementation, where each basic model element is represented by its own type. On top of that, it also sets the course for easily defining element-specific attributes in an implementation. If an implementation does not require these model-specific basic types though, they may collapse into one basic type or may even be implemented by platform-specific basic data types like integer or string.

The RBAC model (Appendix A.3) defines five basic elements, on which all other model components are based: users $u \in U$, sessions $s \in S$, roles $r \in R$, objects $o \in O$, and operations $op \in OP$. Applying the first step to the RBAC model results in five basic types, which we call `User_T`, `Session_T`, `Role_T`, `Object_T`, and `Operation_T`. The operators of these ADTs naturally depend on the model; they are derived from the model-independent basic types by the formal rule system as shown below.

The next step is to apply the formal rule system to the model-specific component algebras. Afterwards, all derived TCB function sets are united and each TCB function is attached to the ADT that it belongs to. Result is a redundant-free set of TCB functions that is described by ADT operators. We demonstrate this step by means of a collection of class diagrams. For each component algebra, we gradually enhance the previous class diagram to illustrate the added ADT operators and the ADTs' interrelations. This notation is sufficient since it is not necessary to define the semantics of the resulting operators again; we have already provided the operators' semantics in Section 4.4.4.

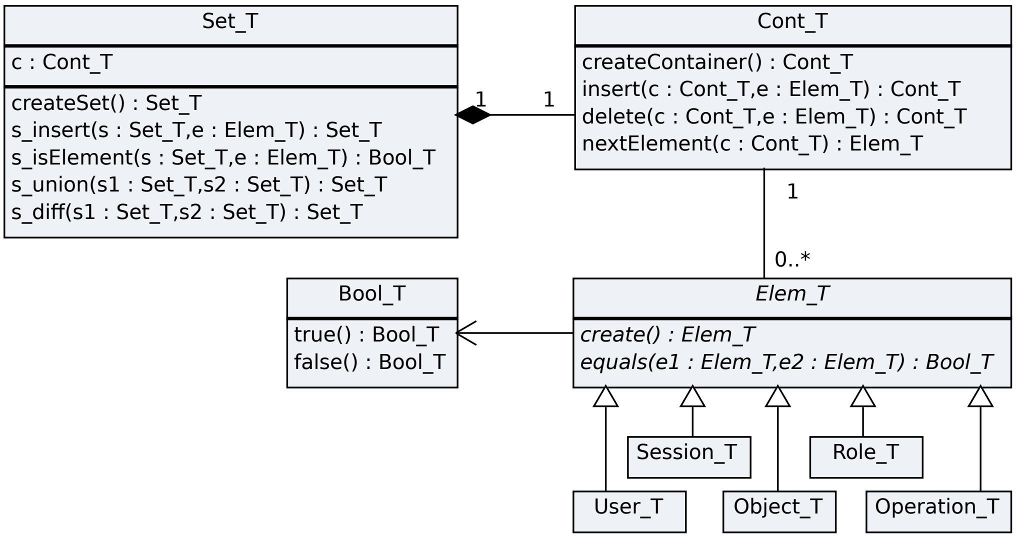
Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Set_T,e : Elem_T) : Set_T
s_isElement(s : Set_T,e : Elem_T) : Bool_T
s_union(s1 : Set_T,s2 : Set_T) : Set_T
s_diff(s1 : Set_T,s2 : Set_T) : Set_T
1
1
Cont_T
createContainer() : Cont_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
nextElement(c : Cont_T) : Elem_T
1
0..*
Bool_T
true() : Bool_T
false() : Bool_T
Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T
Session_T
Role_T
User_T
Object_T
Operation_T

Figure 4.24: Derived ADTs for Algebra *Set* for RBAC HIS Model

We start with applying the rule system to the model-specific component algebra *Set*. *Set* defines five operations, from which TCB functions are derived by all *Set*-based rules besides the 6. rule. The 1. rule adds the operator `createContainer()`. The 2. rule derives the functions `next()`, `create()`, `insert()`, `equals()`, `false()`, and `true()`. Rule 3 and 4 do not lead to any new TCB functions, since all required functions are already contained in the derived function set $\mathcal{O}_S$. The 5. rule finally adds the function `delete()`. Result is a TCB function set $\mathcal{O}_S$ that contains eight functions defined on the ADTs `Cont_T`, `Elem_T`, and `Bool_T`. Besides that, the ADT `Set_T`, which is directly produced by *Set*, contains five interface functions as defined by the algebra: `createSet()`, `s_insert()`, `s_isElement()`, `s_union()`, and `s_diff()`.

$\mathcal{O}_S := \{$
```
Cont_T createContainer(),
Elem_T next(Cont_T c),
Cont_T insert(Cont_T c, Elem_T e),
Cont_T delete(Cont_T c, Elem_T e),
Elem_T create(),
Bool_T equals(Elem_T e1, Elem_T e2),
Bool_T false(),
Bool_T true()}
```

We now have to combine the model-specific basic types with the derived ADT functions. The model-independent basic type `Elem_T` implements two functions `create()` and `equals()`. Since model-dependent basic types refine model-independent basic types, all model-specific basic types have to provide these two functions. For this reason, the class digram shown in Figure 4.24 models `Elem_T` as abstract class, from which all model-specific basic types `User_T`, `Session_T`, `Role_T`, `Object_T`, and `Operation_T` are derived, each of which implementing the ADT operators `create()` and `equals()` for their needs. However, since both TCB functions have the same semantics for each type (otherwise they would have different names), they are contained in the TCB function set only once, thus avoiding functional redundancies.

$\mathcal{O}_R := \{$
```
Cont_T createContainer(),
Elem_T next(Cont_T c),
Elem_T create(Dom_T d, Tar_T t),
Cont_T insert(Cont_T c, Elem_T e),
Cont_T delete(Cont_T c, Elem_T e),
Bool_T equals(Dom_T d1, Dom_T d2),
Dom_T getFirst(),
Tar_T getSecond(),
Bool_T false(),
Bool_T true(),
Bool_T and(Bool_T b1, Bool_T b2),
Set_T createSet(),
Set_T s_insert(Set_T s, Elem_T e)}
```

In the following, we apply the same procedure to the model-specific component algebra *Relation*, defining a set of six operations. The derived function set $\mathcal{O}_R$ is composed by the functions produced from rules 1-6, which require three additional ADTs `TElem_T`, `Dom_T`, and `Tar_T`. As can be seen, all functions defined on `Cont_T` and `Elem_T` have already been derived from *Set*. `Bool_T` defines an additional operator `and()`. Rule 6 requires the interface functions `createSet()` and `s_insert()`, and the basic operators to implement them. These operators (`createContainer()`, `next()`, `insert()` and so on) are already contained in $\mathcal{O}_R$ and do not need to be added again. Finally, the interface functions defined on `Rel_T` are `createRelation()`, `r_insert()`, `r_isElement()`, `r_union()`, `r_diff()`, and `r_domainSub()`.

Adding the derived functions to the ones derived from *Set* results in the class diagram shown in Figure 4.25. The ADTs `Cont_T`, `Set_T`, and `Elem_T` have already been derived from *Set* and are reused, which is indicated by the namespace `RBAC-SET`. The class `TElem_T` is derived from the abstract class `Elem_T`. Besides implementing the abstract functions `create()` and `equals()`, it implements tuple-specific operations called `getFirst()` and `getSecond()`. Model-specific basic data types are now not only derived from `Elem_T`, but also from `Dom_T` and `Tar_T`. `Role_T` is is derived from `Tar_T` since the base set of the *Target* sort of *Relation* is defined by the set of roles. In contrast, the base set of the sort *Domain* is defined by a unification of the user set and the role set. Thus, the basic types `User_T` and `Role_T` must also be derived from `Dom_T`.

The next step is to derive TCB functions for the model-specific component algebra *Mapping*. Except for rules 4 and 6, all rules defined on *Mapping* are applied. As can been seen, there are no additional functions that have not already been derived from *Relation*. The only difference is the additional ADT `Map_T` and its interface functions `createMapping()`, `mp_init()`, `mp_insert()`, `mp_isElement()`, `mp_union()`, `mp_diff()`, `mp_domainSub()`, `mp_rangeSub()`, `mp_getMapping()`, and `mp_override()`.

From this it follows that the resulting class diagram (Figure 4.26) contains one new ADT `Map_T`. All other ADTs and their operators have already been derived before and are reused (namespaces `RBAC-SET` and `RBAC-RELATION`). However, there are two significant differences: (i) here, the model-independent basic type `Session_T` is not only derived from `Elem_T` but also from `Dom_T`. The reason is that the base set of the *Domain* sort of *Mapping* is defined by the model's session set. (ii) `Set_T` is derived from `Tar_T` since the base set of sort *Target* is defined by unifying the model's user set and the power set of the model's role set. The latter is supported by `Set_T`, which is implemented by `Cont_T`, consisting of arbitrary objects of

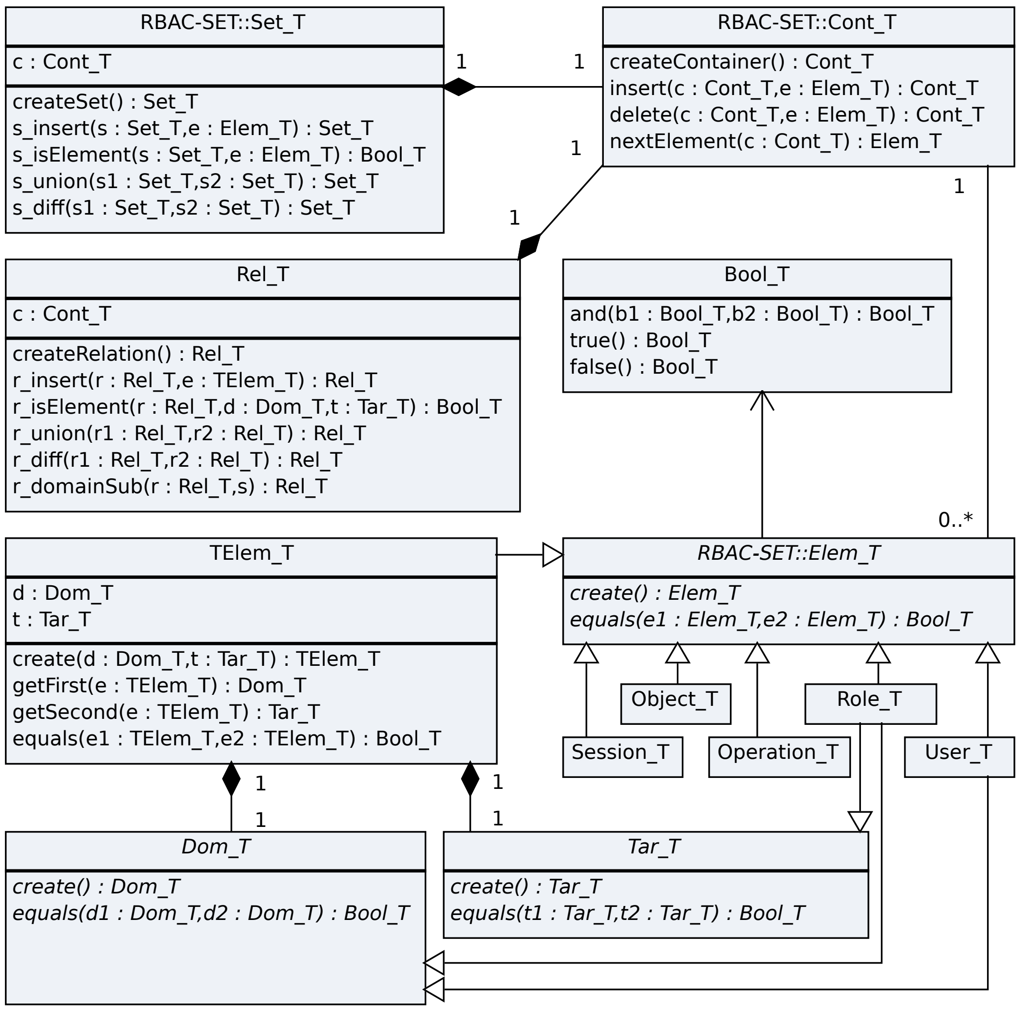
RBAC-SET::Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Set_T,e : Elem_T) : Set_T
s_isElement(s : Set_T,e : Elem_T) : Bool_T
s_union(s1 : Set_T,s2 : Set_T) : Set_T
s_diff(s1 : Set_T,s2 : Set_T) : Set_T
RBAC-SET::Cont_T
createContainer() : Cont_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
nextElement(c : Cont_T) : Elem_T
Rel_T
c : Cont_T
createRelation() : Rel_T
r_insert(r : Rel_T,e : TElem_T) : Rel_T
r_isElement(r : Rel_T,d : Dom_T,t : Tar_T) : Bool_T
r_union(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_diff(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_domainSub(r : Rel_T,s) : Rel_T
Bool_T
and(b1 : Bool_T,b2 : Bool_T) : Bool_T
true() : Bool_T
false() : Bool_T
TElem_T
d : Dom_T
t : Tar_T
create(d : Dom_T,t : Tar_T) : TElem_T
getFirst(e : TElem_T) : Dom_T
getSecond(e : TElem_T) : Tar_T
equals(e1 : TElem_T,e2 : TElem_T) : Bool_T
RBAC-SET::Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T
Object_T
Role_T
Session_T
Operation_T
User_T
Dom_T
create() : Dom_T
equals(d1 : Dom_T,d2 : Dom_T) : Bool_T
Tar_T
create() : Tar_T
equals(t1 : Tar_T,t2 : Tar_T) : Bool_T
1
1
1
1
1
0..*
1
1
1
1

Figure 4.25: Derived ADTs for Algebras *Set* and *Relation* for RBAC HIS Model

type `Elem_T`, from which `Role_T` is derived.

It remains to apply the rule system to the algebra *Matrix*. Here, rules 1,2,3, and 5 can be applied, resulting in the TCB function set $\mathcal{O}_M$. As can be seen, three additional ADTs are necessary, which we call `TrElem_T`, `Row_T`, and `Col_T`. All other ADTs are already derived in the previous steps, including `Set_T` with its operators `createSet()` and `s_insert()` (rule 2). Besides that, `Matrix_T` contains four interface functions `createMatrix()`, `m_init()`, `m_insert()`, and `m_cellContent()` as specified by the algebra *Matrix*.

$\mathcal{O}_M := \{$
```
Cont_T createContainer(),
Elem_T next(Cont_T c),
TrElem_T create(Dom_T d, Tar_T t, Set_T s),
Cont_T insert(Cont_T c, Elem_T e),
Bool_T equals(Row_T r1, Row_T r2),
Row_T getFirst(),
Col_T getSecond(),
Tar_T getThird(),
Bool_T false(),
Bool_T true(),
Bool_T and(Bool_T b_1, Bool_T b_2),
Set_T createSet(),
Set_T s_insert(Set_T s, Elem_T e)}
```

The last step is to unify all derived TCB function sets as done in Figure 4.27. As can be seen, `Matrix_T`, `TrElem_T`, `Row_T`, and `Col_T` are the only new classes; all other have already been derived before and are reused as shown by namespaces `RBAC-SET`, `RBAC-RELATION`, and `RBAC-MAPPING`. Like `TElem_T`, `TrElem_T` is also derived from the abstract `Elem_T`. Besides the functions `create()` and `equals()` that are defined by `Elem_T`, `TrElem_T` also implements triple-specific functions called `getFirst()`, `getSecond()`, and `getThird()`. While a pair of `Row_T` and `Col_T` objects represents the domain of a `Matrix_T` object, the target object is constructed by a `Set_T` object. Like `Dom_T` and `Tar_T`, `Row_T` and `Col_T` also implement the functions `create()` and `equals()` in dependence on their objects. As can be seen, we now have four ADTs that have to implement exactly the same functions; however, the functions are still contained in the function set only once. The model-dependent basic type `Object_T` is derived from `Col_T`; `Role_T` is additionally derived from `Row_T`. The reason is that the base sets of the sorts *Row* and *Column* of the *Matrix* algebra are defined by the object and the row set. All other model-specific basic types remain the same.

At this stage of ADT engineering, the entire set of TCB functions to implement the TCB component `RBAC security model functions` is defined. Though by further structuring the TCB functions, we reduce the gap between a TCB's functions and its platform-dependent implementation. For this purpose, we follow the last step of ADT engineering, which converts the core algebras *State* and *ExtensionVector* to the corresponding ADTs `State_T` and `ExtVec_T`. The operations of these ADTs represent the model's primitive actions and clauses, which are implemented by the already derived basic ADT operators.

sorts	State_T
import	Bool_T, User_T, Session_T, Role_T, Set_T, Rel_T, Map_T
operators	createState: Set_T × Set_T × Rel_T × Map_T × Map_T → State_T getUserS: State → Set_T getSessionS: State → Set_T getUAR: State → Rel_T getRolesMap: State → Map_T getUsersMap: State → Map_T addUsers: State_T × Set_T → State_T deleteUsers: State_T × Set_T → State_T createSessions: State_T × Set_T → State_T destroySessions: State_T × Set_T → State_T mapUserSessions: State_T × Set_T × User_T → State_T unmapUserSessions: State_T × Set_T × User_T → State_T assignUserToRoles: State_T × User_T × Set_T → State_T revokeUserFromRoles: State_T × User_T × Set_T → State_T activateRoles: State_T × Session_T × Set_T → State_T deactivateRoles: State_T × Session_T × Set_T → State_T $\mathtt{clause}_{\mathtt{roles}}$: State_T × Session_T × Role_T → Bool_T $\mathtt{clause}_{\mathtt{UA}}$: State_T × Role_T × User_T → Bool_T
variables	$\forall$q : State_T, S, S_1 : Set_T, R : Rel_T, Map, Map_1 : Map_T, u : User_T, c : Cont_T, s : Session_T, r : Role_T

axioms	createState(S,S_1,R,Map,Map_1):= function_of(s_union(), createSet(), r_union(), createRelation(), mp_init())
	addUsers(q,S):= function_of(s_union(), getUserS())
	deleteUsers(q,S):= function_of(s_diff(), getUserS(), r_domainSub(), getUAR(), mp_rangeSub(), getUsersMap())
	createSessions(q,S):= function_of(s_union(), getSessions(), next(), mp_override(), getRolesMap(),mp_insert(), createMapping(), createSet())
	destroySessions(q,S):= function_of(s_diff(), getSessions(), mp_domainSub(), getUsersMap(), getRolesMap())
	mapUserSessions(q,S,u):= function_of(mp_override(), getUsersMap(), mp_insert(), createMapping(), next())
	unmapUserSessions(q,S,u):= mp_diff(getUsersMap(), mp_insert(), createMapping(), next())
	assignUserToRoles(q,u,S):= function_of(r_union(), getUAR(), next(), r_insert(), createRelation())
	revokeUserFromRoles(q,u,S):= function_of(r_diff(), getUAR(), next(), r_insert(), createRelation())
	activateRoles(q,s,S):= function_of(mp_override(), getRolesMap(), mp_insert(), getRolesMap(), next(), s_insert(), createSet())
	deactivateRoles(q,s,S):= function_of(mp_override(), getRolesMap(), mp_diff(), mp_insert(), createMapping())
	$clause_{roles}$(q,s,r,) := function_of(s_isElement(), mp_getMapping(), getRolesMap())
	$clause_{UA}$(q,r,u) := function_of(r_isElement(), getUAR(), r_insert(), createRelation())

Table 4.25: Axioms of ADT `State_T` for RBAC HIS Model

The model-specific ADT `State_T` is shown in Table 4.25. As can be seen, its operator set contains representatives for all operations of the algebra *State*. It imports the sorts `Bool_T`, `User_T`, `Session_T`, `Role_T`, and `Set_T` to define the input and output parameter sorts of its operators. The sorts `Cont_T`, `Rel_T`, and `Map_T` are only imported since their operators are needed to define some of the axioms. However, the axiom set does not define axioms in a the narrower sense. Here, axioms specify which basic ADT operators and interface operators are required to implement the primitive actions and clauses of the models. We therefor define a new operator `function_of()`, which denotes that the semantics of an ADT's operator is specified by a set of interface operators and basic operators. For example, the axiom `addUser(q,S) := function_of(s_union(), getUsers())` states that the functions `s_union()` and `getUSers()` are required to implement the semantics of `addUser()`[8].

It remains to convert the algebra *ExtensionVector* into the corresponding ADT `ExtVect_T`. As can be seen in Table 4.26, the ADT defines exactly the same operators as its algebra.

[8]Note that the axioms only state which of the derived functions are required; however, they do not consider mechanisms of flow control or the operator's composition.

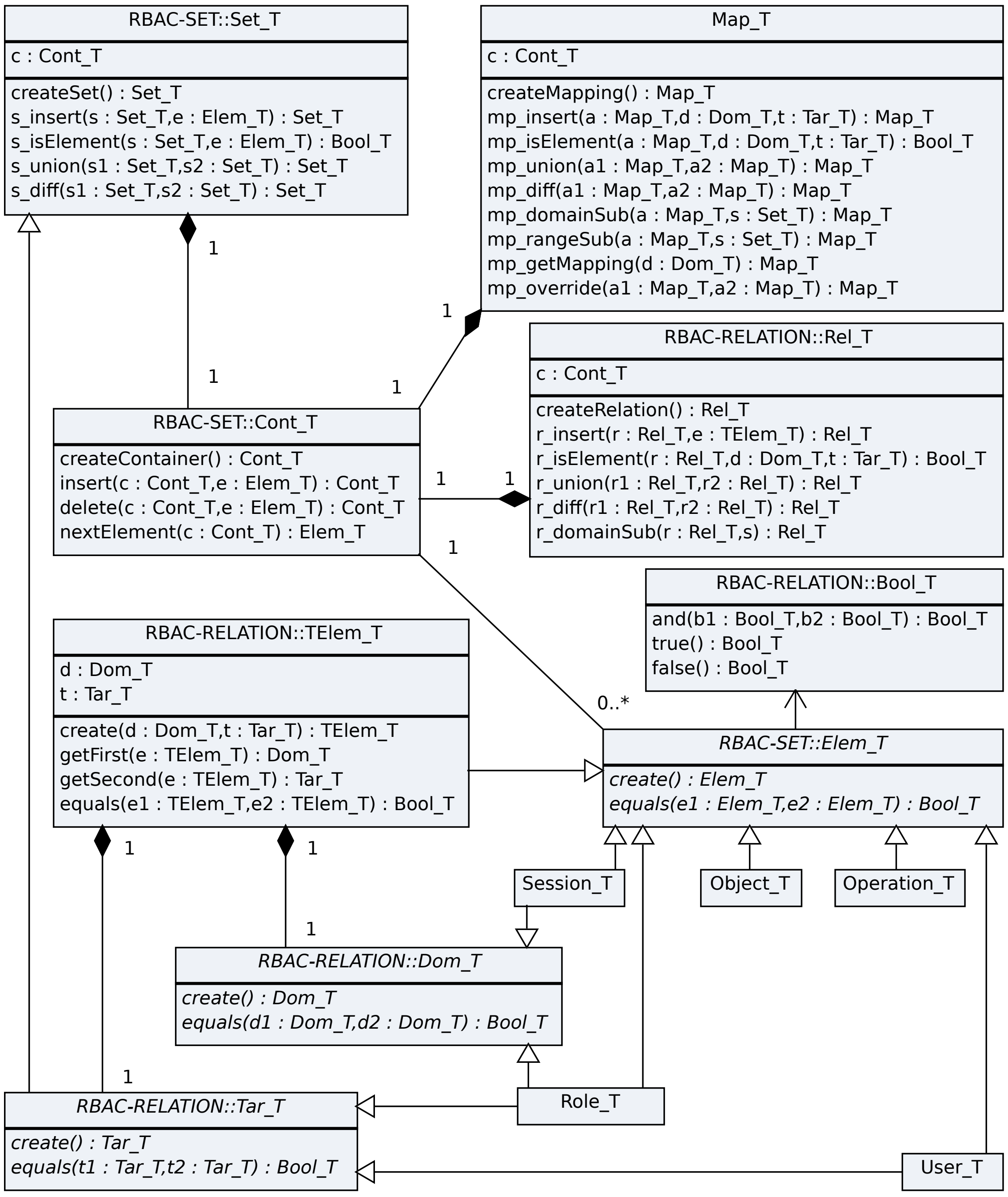
RBAC-SET::Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Set_T,e : Elem_T) : Set_T
s_isElement(s : Set_T,e : Elem_T) : Bool_T
s_union(s1 : Set_T,s2 : Set_T) : Set_T
s_diff(s1 : Set_T,s2 : Set_T) : Set_T
Map_T
c : Cont_T
createMapping() : Map_T
mp_insert(a : Map_T,d : Dom_T,t : Tar_T) : Map_T
mp_isElement(a : Map_T,d : Dom_T,t : Tar_T) : Bool_T
mp_union(a1 : Map_T,a2 : Map_T) : Map_T
mp_diff(a1 : Map_T,a2 : Map_T) : Map_T
mp_domainSub(a : Map_T,s : Set_T) : Map_T
mp_rangeSub(a : Map_T,s : Set_T) : Map_T
mp_getMapping(d : Dom_T) : Map_T
mp_override(a1 : Map_T,a2 : Map_T) : Map_T
RBAC-RELATION::Rel_T
c : Cont_T
createRelation() : Rel_T
r_insert(r : Rel_T,e : TElem_T) : Rel_T
r_isElement(r : Rel_T,d : Dom_T,t : Tar_T) : Bool_T
r_union(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_diff(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_domainSub(r : Rel_T,s) : Rel_T
RBAC-SET::Cont_T
createContainer() : Cont_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
nextElement(c : Cont_T) : Elem_T
RBAC-RELATION::Bool_T
and(b1 : Bool_T,b2 : Bool_T) : Bool_T
true() : Bool_T
false() : Bool_T
RBAC-RELATION::TElem_T
d : Dom_T
t : Tar_T
create(d : Dom_T,t : Tar_T) : TElem_T
getFirst(e : TElem_T) : Dom_T
getSecond(e : TElem_T) : Tar_T
equals(e1 : TElem_T,e2 : TElem_T) : Bool_T
RBAC-SET::Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T
Session_T
Object_T
Operation_T
RBAC-RELATION::Dom_T
create() : Dom_T
equals(d1 : Dom_T,d2 : Dom_T) : Bool_T
RBAC-RELATION::Tar_T
create() : Tar_T
equals(t1 : Tar_T,t2 : Tar_T) : Bool_T
Role_T
User_T
1
1
1
1
1
1
1
1
0..*
1
1
1
1

Figure 4.26: Derived ADTs for Algebras Set, $Relation$, and $Mapping$ for RBAC HIS Model

$\mathcal{O}_{Map} := \{$
```
Cont_T createContainer(),
Elem_T next(Cont_T c),
Elem_T create(Dom_T d, Tar_T t),
Cont_T insert(Cont_T c, Elem_T e),
Cont_T delete(Cont_T c, Elem_T e),
Bool_T equals(Tar_T t1, Tar_T t2),
Dom_T getFirst(),
Tar_T getSecond(),
Bool_T false(),
Bool_T true(),
Set_T createSet(),
Set_T s_insert(Set_T s, Elem_T e)}
```

Again, the axioms describe which interface operators are required to implement the constructor operator and the model's clauses. This is feasible, since the semantics of the ADT's operators is well defined by the corresponding operations of the algebra *ExtensionVector*.

sorts	ExtVec_T
import	Bool_T, Operation_T, Object_T, Role_T, Set_T, Rel_T, Matrix_T
operators	createExtVec: Set_T × Set_T × Set_T × Matrix_T × Rel_T × Rel_T → ExtVec_T getObjectS: → Set_T getOperationS: → Set_T getRoleS: → Set_T getMatrixM: → Matrix_T getRHR: → Rel_T getRER: → Rel_T $\texttt{clause}_{\texttt{m}}$: Operation_T × Role_T × Object_T → Bool_T $\texttt{clause}_{\texttt{RH}}$: Role_T × Role_T → Bool_T $\texttt{clause}_{\texttt{RE}}$: Role_T × Role_T → Bool_T
variables	$\forall S, S_1, S_2, S_3 : \texttt{Set\_T}, R, R_1 : \texttt{Rel\_T}, M : \texttt{Matrix\_T}, o : \texttt{Object\_T}, r, r_1 : \texttt{Role\_T}$, $op : \texttt{Operation\_T}$
axioms	createExtVec(S,S_1,S_2,M,R,R_1):= function_of(s_union(), createSet(), m_insert(), createMatrix(), r_union(), createRelation()) $\texttt{clause}_{\texttt{m}}$(op,r,o) := function_of(s_isElement(), m_cellContent(), getMatrixM()) $\texttt{clause}_{\texttt{RH}}$(r,$r_1$) := function_of(r_isElement(), getRHR()) $\texttt{clause}_{\texttt{RE}}$(r,$r_1$) := function_of(r_isElement(), getRE())

Table 4.26: ADT ExtVec_T for RBAC HIS Policy

The ADTs State_T and ExtVec_T can now be used by the executable RBAC HIC policy to define its state and extension vector, in order to implement its authorization scheme. This will be demonstrated in Section 4.5.3.

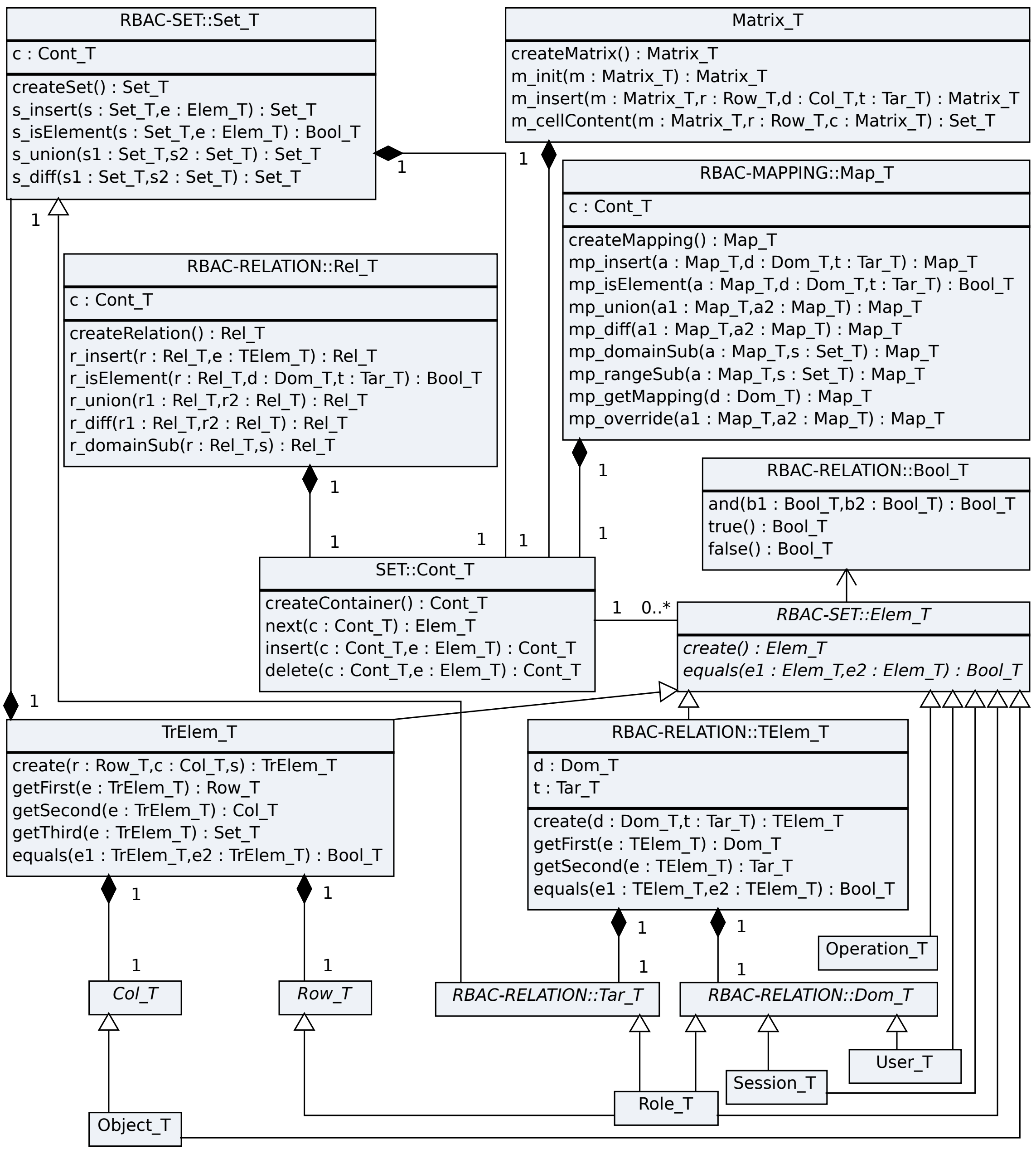
RBAC-SET::Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Set_T,e : Elem_T) : Set_T
s_isElement(s : Set_T,e : Elem_T) : Bool_T
s_union(s1 : Set_T,s2 : Set_T) : Set_T
s_diff(s1 : Set_T,s2 : Set_T) : Set_T
Matrix_T
createMatrix() : Matrix_T
m_init(m : Matrix_T) : Matrix_T
m_insert(m : Matrix_T,r : Row_T,d : Col_T,t : Tar_T) : Matrix_T
m_cellContent(m : Matrix_T,r : Row_T,c : Matrix_T) : Set_T
RBAC-MAPPING::Map_T
c : Cont_T
createMapping() : Map_T
mp_insert(a : Map_T,d : Dom_T,t : Tar_T) : Map_T
mp_isElement(a : Map_T,d : Dom_T,t : Tar_T) : Bool_T
mp_union(a1 : Map_T,a2 : Map_T) : Map_T
mp_diff(a1 : Map_T,a2 : Map_T) : Map_T
mp_domainSub(a : Map_T,s : Set_T) : Map_T
mp_rangeSub(a : Map_T,s : Set_T) : Map_T
mp_getMapping(d : Dom_T) : Map_T
mp_override(a1 : Map_T,a2 : Map_T) : Map_T
RBAC-RELATION::Rel_T
c : Cont_T
createRelation() : Rel_T
r_insert(r : Rel_T,e : TElem_T) : Rel_T
r_isElement(r : Rel_T,d : Dom_T,t : Tar_T) : Bool_T
r_union(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_diff(r1 : Rel_T,r2 : Rel_T) : Rel_T
r_domainSub(r : Rel_T,s) : Rel_T
RBAC-RELATION::Bool_T
and(b1 : Bool_T,b2 : Bool_T) : Bool_T
true() : Bool_T
false() : Bool_T
SET::Cont_T
createContainer() : Cont_T
next(c : Cont_T) : Elem_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
RBAC-SET::Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T
1
0..*
TrElem_T
create(r : Row_T,c : Col_T,s) : TrElem_T
getFirst(e : TrElem_T) : Row_T
getSecond(e : TrElem_T) : Col_T
getThird(e : TrElem_T) : Set_T
equals(e1 : TrElem_T,e2 : TrElem_T) : Bool_T
RBAC-RELATION::TElem_T
d : Dom_T
t : Tar_T
create(d : Dom_T,t : Tar_T) : TElem_T
getFirst(e : TElem_T) : Dom_T
getSecond(e : TElem_T) : Tar_T
equals(e1 : TElem_T,e2 : TElem_T) : Bool_T
Operation_T
Col_T
Row_T
RBAC-RELATION::Tar_T
RBAC-RELATION::Dom_T
User_T
Session_T
Role_T
Object_T

Figure 4.27: Derived ADTs for Algebras *Set*, *Relation*, *Mapping*, and *Matrix* for RBAC HIS Model

4.5.2 Interceptor

To complete the functional range of the policy-depend RTE, we have to specify the functions of the remaining component – the `interceptor`. For this purpose, this section derives the interface functions of the `interceptor` from the RBAC HIS policy. Note that the resulting set of interface functions basically serves as an example for the specification of an `interceptor`, for which a given policy is suitable by implementing the `interceptor`'s interface functions. However, the general approach is not to derive the interface of the `interceptor` from the system's policies, but from the OS and its applications to be protected. In this case, it remains to analyze whether a given policy implements them.

As described in Section 4.4.3, the interface functions of the `interceptor` component must match the authorization scheme of a policy's model instance. The authorization scheme of the RBAC HIS policy (Appendix B.2) consists of 16 commands, 15 of which are state-modifying commands and one is an exemplary non-state-modifying command. This leads to 16 interface functions as shown below. For example, the command

```
$\delta$(q, (createUser $(x_s, x_u)$) ::=
    **if** $(cond_{core}(q, x_s, `U_o`, `update`))$
    **then**
        addUsers $(q, \{x_u\})$
    **end if.**
```

requires two input parameters x_s and x_u where the indexes s and u indicate that the first parameter must be a session $s \in S$ and the second one must be a user $u \in U$. This results in the interface function `public void createUser(Session_T s, User_T u)` with `Session_T` and `User_T` being model-specific basic types as defined in Section 4.5.1. Since this function modifies the state of the policy, it does not have a return parameter; only non-state-modifying commands return the result of their conditions by type `Bool_T`. Note that the names of the commands of a policy's authorization scheme and the interface functions of the `interceptor` must not be identical like in the example; only identical input parameter types are important for the `interceptor`'s mapping.

```
public void createUser(Session_T s, User_T u),
public void destroyUser(Session_T s, User_T u),
public void assignRole(Session_T s, User_T u, Role_T r),
public void revokeRole(Session_T s, User_T u, Role_T r),
public void login(Session_T s, User_T u),
public void logout(Session_T s),
public void activateRole(Session_T s, Role_T r),
public void deactivateRole(Session_T s, Role_T r),
public void deactivateRole(Session_T s, Role_T r),
public void assignReferredDoctorRole(Session_T s, User_T u),
public void revokeReferredDoctorRole(Session_T s, User_T u),
public void assignPatientRole(Session_T s, User_T u,
public void revokePatientRole(Session_T s, User_T u),
public void assignMedicalTeamRole(Session_T s, User_T u),
public void revokeMedicalTeamRole(Session_T s, User_T u),
public Bool_T view(Session_T s, Object_T o)
```

As shown in Appendix B.2, all input parameters of the authorization scheme's commands are defined by basic elements of the model, i.e., users, sessions, roles, and objects. This results in that the input parameters of all functions of the `interceptor` are specified by the model-specific basic types `User_T`, `Session_T`, `Role_T`, and `Object_T`, which represent these primitive types. This is not necessarily so for the authorization schemes of arbitrary policies; input parameters may also be of types `Set_T`, `Rel_T`, `Map_T`, or `Matrix_T`.

At this stage, the functional perimeter for a policy-dependent RTE that is tailored to enforce the RBAC HIS policy is specified. It now remains to show how an `executable security policy` interacts with the TCB's `interceptor` and the `security model functions` component that is derived from its model. This will be demonstrated by means of the `executable RBAC HIS policy` in the following section.

4.5.3 Executable RBAC HIS Policy

The goal of this section is to discuss the design of an `executable security policy` that can be enforced by a policy-dependent RTE. To this end, we focus on the interrelations of the `executable RBAC HIS policy` with the RTE's components `interceptor` and `RBAC security model functions`.

The main responsibility of an `executable security policy` is to implement its authorization scheme. For this purpose, it requires state and extension vector of the model instance as well as the semantics of the primitive actions and clauses. Since all of this is required by any policy, we have designed the RTE in such a way that it provides not only the semantical implementation of the primitive actions and clauses but also data structures for a policy's state and extension vector. The only things that remain for a policy to do are to initialize its state and extension vector according to q_0 and E of its model instance and to implement the commands of its authorization scheme. Besides that, an `executable policy` also has to manage its `threads`, `transactions`, and `TPS` that are provided by the policy-independent RTE; however, this is out of the scope of this section.

Figure 4.28 shows how the `executable RBAC HIS policy` is embedded into a policy-dependent RTE that is tailored to support and enforce this policy. Due to readability reasons, we have illustrated only the main ADTs; for more details refer to the class diagrams in Section 4.5.1. As can be seen, the `RBAC HIS policy` provides exactly the same functions as the `interceptor`. The latter acts as a proxy and calls the appropriate policy function whenever a policy request is received. As already discussed earlier, the function sets of the `interceptor` and an `executable security policy` do not necessarily need to be identical; in case a TCB enforces multiple policies at the same time, merely the unification of all policy functions must be equal to the functions of the `interceptor`.

As already motivated, the `executable RBAC HIS policy` existentially depends on its state and extension vector. `State_T` defines the state of the RBAC HIS model, which is existentially dependent on those ADTs that implement its components: `State_T` consists of two sets representing the user and session set, a relation for representing UA, and two mappings for *user* and *roles*. `ExcVec_T` defines the data structure for a model's extension vector using the ADTs `Set_T`, `Rel_T`, and `Matrix_T`. To implement the commands of the authorization scheme, a policy can call the operators provided by `State_T` and `ExcVec_T`. For example, the command `createUser(s,u)` can be implemented as shown by Algorithm 2; all other commands are implemented analogously.

As can be seen, the policy also requires the interface functions of the ADTs `Set_T` and `Bool_T`. Moreover, `next()` of `Cont_T` is also necessary for iterating the static set of roles `r`

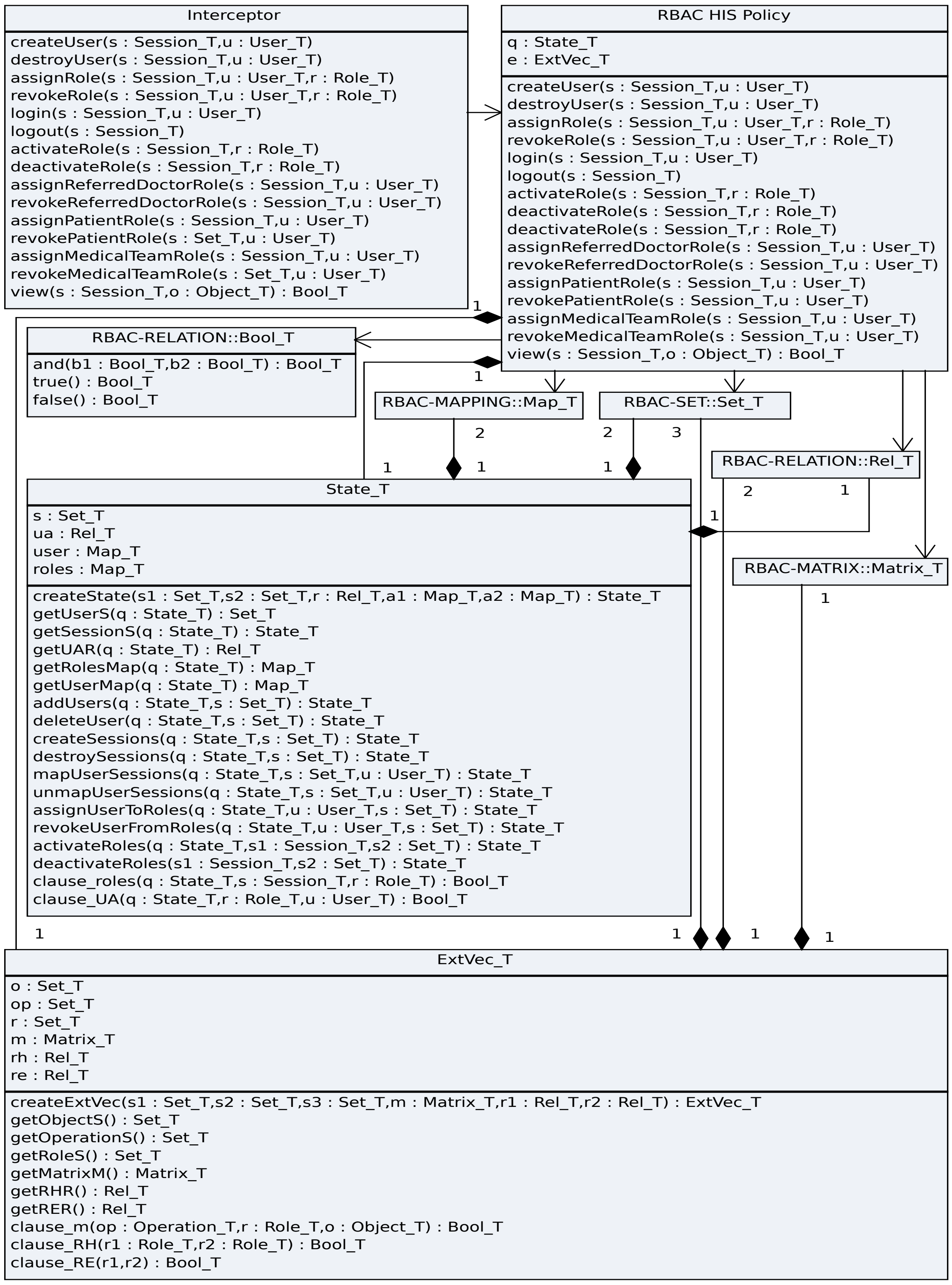
Interceptor
createUser(s : Session_T,u : User_T)
destroyUser(s : Session_T,u : User_T)
assignRole(s : Session_T,u : User_T,r : Role_T)
revokeRole(s : Session_T,u : User_T,r : Role_T)
login(s : Session_T,u : User_T)
logout(s : Session_T)
activateRole(s : Session_T,r : Role_T)
deactivateRole(s : Session_T,r : Role_T)
assignReferredDoctorRole(s : Session_T,u : User_T)
revokeReferredDoctorRole(s : Session_T,u : User_T)
assignPatientRole(s : Session_T,u : User_T)
revokePatientRole(s : Set_T,u : User_T)
assignMedicalTeamRole(s : Session_T,u : User_T)
revokeMedicalTeamRole(s : Set_T,u : User_T)
view(s : Session_T,o : Object_T) : Bool_T
RBAC HIS Policy
q : State_T
e : ExtVec_T
createUser(s : Session_T,u : User_T)
destroyUser(s : Session_T,u : User_T)
assignRole(s : Session_T,u : User_T,r : Role_T)
revokeRole(s : Session_T,u : User_T,r : Role_T)
login(s : Session_T,u : User_T)
logout(s : Session_T)
activateRole(s : Session_T,r : Role_T)
deactivateRole(s : Session_T,r : Role_T)
deactivateRole(s : Session_T,r : Role_T)
assignReferredDoctorRole(s : Session_T,u : User_T)
revokeReferredDoctorRole(s : Session_T,u : User_T)
assignPatientRole(s : Session_T,u : User_T)
revokePatientRole(s : Session_T,u : User_T)
assignMedicalTeamRole(s : Session_T,u : User_T)
revokeMedicalTeamRole(s : Session_T,u : User_T)
view(s : Session_T,o : Object_T) : Bool_T
1
RBAC-RELATION::Bool_T
and(b1 : Bool_T,b2 : Bool_T) : Bool_T
true() : Bool_T
false() : Bool_T
1
RBAC-MAPPING::Map_T
RBAC-SET::Set_T
2
2
3
1
1
1
RBAC-RELATION::Rel_T
2
1
1
State_T
s : Set_T
ua : Rel_T
user : Map_T
roles : Map_T
createState(s1 : Set_T,s2 : Set_T,r : Rel_T,a1 : Map_T,a2 : Map_T) : State_T
getUserS(q : State_T) : Set_T
getSessionS(q : State_T) : State_T
getUAR(q : State_T) : Rel_T
getRolesMap(q : State_T) : Map_T
getUserMap(q : State_T) : Map_T
addUsers(q : State_T,s : Set_T) : State_T
deleteUser(q : State_T,s : Set_T) : State_T
createSessions(q : State_T,s : Set_T) : State_T
destroySessions(q : State_T,s : Set_T) : State_T
mapUserSessions(q : State_T,s : Set_T,u : User_T) : State_T
unmapUserSessions(q : State_T,s : Set_T,u : User_T) : State_T
assignUserToRoles(q : State_T,u : User_T,s : Set_T) : State_T
revokeUserFromRoles(q : State_T,u : User_T,s : Set_T) : State_T
activateRoles(q : State_T,s1 : Session_T,s2 : Set_T) : State_T
deactivateRoles(s1 : Session_T,s2 : Set_T) : State_T
clause_roles(q : State_T,s : Session_T,r : Role_T) : Bool_T
clause_UA(q : State_T,r : Role_T,u : User_T) : Bool_T
RBAC-MATRIX::Matrix_T
1
1
1
1
1
ExtVec_T
o : Set_T
op : Set_T
r : Set_T
m : Matrix_T
rh : Rel_T
re : Rel_T
createExtVec(s1 : Set_T,s2 : Set_T,s3 : Set_T,m : Matrix_T,r1 : Rel_T,r2 : Rel_T) : ExtVec_T
getObjectS() : Set_T
getOperationS() : Set_T
getRoleS() : Set_T
getMatrixM() : Matrix_T
getRHR() : Rel_T
getRER() : Rel_T
clause_m(op : Operation_T,r : Role_T,o : Object_T) : Bool_T
clause_RH(r1 : Role_T,r2 : Role_T) : Bool_T
clause_RE(r1,r2) : Bool_T

Figure 4.28: Integration of `Executable RBAC HIS Policy` in Policy-dependent RTE

contained by `ExtVec_T`. Here, we have added an interface function `next()` to `Set_T`, which

Algorithm 2: Implementation of `createUser(s,u)`

```
Input: e: Ext_T, q:State_T
function createUser(Session_T s, User_T u)
begin
    Role_T r_1;
    Role_T r_2;
    for r_1:= e.getRoleS().next() do
        for r_2:= e.getRoleS().next() do
            Bool_T b := and(clause_RH(r_1,r_2),clause_roles(q,s,r_1));
            if and(b,clause_m('update',r_2,'U_o')) then
                Set_T us := s_insert(createSet(),u);
                q := q.addUsers(q,us);
```

serves as wrapper for the function `next()` defined on `Cont_T`. This is feasible since we have not actually added a new function to the TCB but only a wrapper function to access an already contained function more easily.

This section has illustrated the design of the `executable RBAC HIS policy`. By this means, it has shown that the policy-dependent RTE tailored to enforce the RBAC HIS policy provides everything that the `executable RBAC HIS security policy` requires. On top of that, this section has demonstrated that it is straightforward to integrate an `executable security policy` into an existing policy-dependent RTE by applying its data structures (ADTs) and their functions (operators) to implement the policy's authorization scheme.

4.6 Conclusion

Today's commodity policy-controlled operating systems share the ambition to support a wide range of security policies, rendering their policy runtime environment (RTE) and enforcement mechanisms large, complex, and expensive. As a consequence, their TCBs are characterized by a large functional perimeter that is hard to identify.

This dissertation aims at engineering a policy RTE that supports only those policies that are actually present in a TCB. The goal is to exactly determine a TCB's functional perimeter by exploiting causal dependencies between policies and TCB functions. Causal TCBs then contain only those functions that are necessary to establish, enforce, and protect the policies that are present in a TCB.

This section has developed a functional TCB design along with a TCB engineering method. The functional design of causal TCBs is similar to core-based models; it contains a common TCB component – the policy-independent RTE – and a policy-specific component – the policy-depend RTE. Causal TCBs are engineered by tailoring the policy-dependent RTE due to causal dependencies between policies and TCB functions. We have identified the functional perimeter of the policy-independent RTE by motivating each functional component and the interface it provides. Additionally, an engineering method has been developed for tailoring the policy-dependent RTE. Based on an algebraic approach, TCB functions in ADT notation are derived by a rule system that formalizes said causal dependencies. We have demonstrated the TCB engineering approach by engineering a policy-dependent RTE for the RBAC HIS policy.

The engineering method for a policy-dependent RTE is based on a sound formal foundation. As a consequence, it can be applied to develop an automated TCB composition tool that composes the functional perimeter of causal TCBs. Based on design principles that are inspired by microkernel architectures, the derived functional perimeter for the policy-independent RTE is redundancy-free. The derived function set of the policy-dependent RTE is also nonredundant, since we unify all derived function sets. Moreover, by reducing the semantic gap between formal security models and and implementation-oriented TCB functions, the algebraic approach contributes to a complete functional perimeter with respect to the enforced policies.

5 Specification Engineering

I am among those who think that science has great beauty. A scientist in his laboratory is not only a technician: he is also a child placed before natural phenomena which impress him like a fairy tale.

Marie Curie,
Madame Curie, by Ève Curie Labouisse, 1937

After having engineered a causal TCB, the next step of model-based security engineering is to generate an implementation based on a TCB's identified functions. While the implementation of the policy-independent RTE is to be accomplished once, the policy-dependent RTE has to be implemented individually for each application scenario. It is significant for both, however, that completeness and consistency of their implementations with respect to the identified TCB functions have to be ensured.

In general, implementations that are amenable to correctness and consistency validation are achieved by formal methods [250]. These rewrite a system specification in a formal mathematical specification, which then allows for formal validation techniques, and for generating an implementation that is also amenable to verification.

Specification engineering, i.e., rewriting a system specification as formal specification, has already been applied successfully for validating the correctness of OS kernels, e.g [57,58,137]. Consequently, this approach can also be applied to the policy-independent RTE to produce a specification that serves as basis for the implementation. This is a nonrecurring effort, which significantly contributes to the implementation correctness of the policy-independent RTE of all causal TCBs.

In contrast, formally specifying the policy-dependent RTE of causal TCBs is a repeating process, since we have to specify it for each application scenario based on the derived TCB functions. Hence, method and tool support are significant.

The goal of this section is to show that implementing causal TCBs can indeed be supported by methods and tools. In doing so, we focus on method support for implementing the policy-dependent RTE, since this is a frequently repeating process. To this end, this section develops a specification method that allows for rewriting the functions of a policy-dependent RTE in a formal TCB specification that is amenable to correctness and consistency validation with respect to the enforced security policies. Based on such specifications, compiler-like tools can be developed, which allow for an automated generation of the required source code of a TCB's policy-dependent RTE.

On this account, Section 5.1 briefly introduces the fundamentals of specification engineering to provide the basis for the following sections. A TCB specification has to meet a couple of requirements to serve as basis for code generation. Section 5.2 discusses these requirements before Section 5.3 presents approaches to meet them. The specification method is then presented in Section 5.4.

5.1 Specification Fundamentals

The goal of this section is to present the fundamentals of specification engineering. On this account, it first discusses its connection to model-based security engineering before introducing the required terminology.

Figure 5.1 illustrates how the process of specification engineering is embedded in model-based security engineering: based on a policy in core notation and the derived TCB functions in ADT notation, a formal TCB specification is developed, which serves as basis for tool-supported validation techniques and as compiler input to generate the source code of the functional component `security model functions`. Output is a set of classes, for instance written in *C++* or *Java*, that implement the functions of a policy-dependent RTE and that can be used to implement the authorization scheme of an `executable security policy`. Afterwards, these classes must embedded in a security architecture that implement the functions of the policy-independent RTE. Thereby, specification engineering significantly increases the efficiency of implementing causal TCBs, which is relevant for unlocking causal TCB engineering for real-world application scenarios.

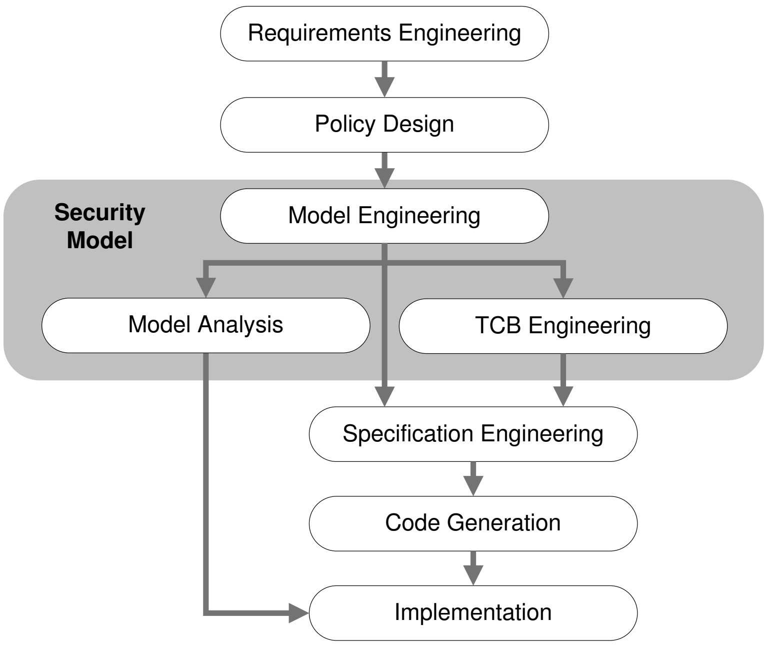

Figure 5.1: Specification Engineering

The basis of specification engineering is a universal specification method that can be applied to all security models in core notation. Input is a set of ADTs that is derived from a specific core-based model. Even though the semantics of the ADT operators is given by the mapping $\beta : \mathscr{P}(\mathcal{O}) \rightarrow \mathscr{P}(\mathcal{U}_O)$, which maps a set of ADT operators to a universe operation (Section 4.4.2.2), the core-based model is also input of the specification method, since the model notation is closer to a formal specification. Output is a formal specification for a sequential, state-based system, from which a platform-specific implementation of a policy-dependent RTE and of an `executable security policy` can be generated.

In general, a state-based specification contains a state machine with a *state* that is described by a set of typed variables x_i[1]. Each x_i is assigned a value $\mathsf{v}_i(c)$; the set of values forms the machine's *initial state*. Valid assignments of these variables are expressed by *invariants* $\mathsf{I}_i(\mathsf{x}, \mathsf{c})$ (statements in predicate logic). If a specification only produces states that do not violate the invariants, it is considered correct; to proof a specification's correctness, *theorems* $\mathsf{J}_i(\mathsf{x}, \mathsf{c})$ are defined to be validated against the invariants. State transitions are specified by a set of *operations* $\mathsf{o}_i : [\mathsf{p}_i, \mathsf{pre}_i, \mathsf{post}_i]$. Each operation consists of typed parameters p_i, a precondition $\mathsf{pre}_i(\mathsf{p}, \mathsf{x}, \mathsf{c})$ that guards its execution, and a postcondition $\mathsf{post}_i(\mathsf{x}', \mathsf{p}, \mathsf{x}, \mathsf{c})$ that defines the operation's semantics by making the state transition from state x to the subsequent state x'.

We have chosen *Event-B* [6, 72] as specification platform. The reasons are its expressive power that allows for modular specifications and the sophisticated tool support by means of the *Rodin* development platform [7, 72].

In order that TCB specifications in Event-B can serve as basis for a TCB's implementation, they have to meet a range of requirements, which we subsequently discuss.

5.2 Requirements for TCB Specifications

Based on the responsibility of TCBs to correctly enforce policies and on the goal of formal specifications to be amenable to validation, we have identified the following requirements for TCB specifications [178]:

1. *Completeness*: A TCB specification must be complete with respect to the security model instance and hence the derived ADT operators. Consequently, there must be a homomorphism between the instance and the specification such that any input and any state of the instance can be mapped to an input and a state of the specification.

2. *External consistency*: A TCB specification must be consistent regarding the model instance. That means, we have to map each input and each state of the specification to an input and a state of the instance. Under consideration of the first requirement, we thus have to build an isomorphism between the instance and the specification.

3. *Inner consistency*: A TCB specification must be self-consistent. To reason about a specification's self-consistency, a set of proof obligations can be defined. Each proof obligation is a statement in predicate logic, which has to be fulfilled for self-consistency. Recent work such as [5,6,8,55,128] has specified different proof obligation types and has developed approaches how to define proof obligations in dependence on a specifications' structure. Based on this work, [178] has identified the following proof obligation types as relevant for ensuring the inner consistency of TCB specifications: well-definedness, type correctness, consistency of axioms, validity of theorems, consistency of invariants, operations, and the initial state, as well as the activation of operations.

4. *Reusability*: To improve the efficiency of specification engineering, TCB specifications should be reused. Here, an approach similar to core-based model engineering should be considered: by building an inheritance hierarchy of specifications (model core, security model, model instance), creating the specification of a security model equals tailoring the specification of the model core. The specification of a model instance then is a refinement of the model's specification that considers the specific instance properties such as its authorization scheme.

[1]For better readability, all components of a TCB specification will be written in sans-serif style.

5. *Small size and low complexity*: To reduce the error-proneness of specification engineering and hence a TCB's implementation, a TCB specification must be of small size and low complexity. On the other hand, Boswell [40] discusses that dedicated specification redundancy contributes to error detection and correction. At first sight, this is in conflict with the requirement, since redundancy quantitatively increases a specification. However, under consideration of its structure, documentation, and provable equivalence, dedicated redundancy does not increase the complexity at all, but in fact contributes to avoiding specification errors.

5.3 TCB Specification Approach

This section presents specification approaches that tackle the requirements of Section 5.2 and thus build the foundation of the specification method.

The requirements *completeness* and *external consistency* are met be a methodical approach. By listing all elements of a core-based model instance as done in Appendix B, we can stepwise rewrite a model instance as formal specification. This leads to a complete specification with a detailed case-by-case specification and a detailed documentation of all rewriting sub-steps.

External consistency of a formal specification is ensured both during and after the process of specification engineering. This is done by random sampling with the help of an automated animator tool: based on the initial state of a model instance, we execute the operations of the state-based system and compare the execution results to the results of symbolically executing the model instance. To this end, we derive test cases from the model instance, its application scenario, and the components of the formal specification.

To ensure a specification's *inner consistency*, formal validation techniques are applied. Here, we employ the tools of the *Rodin* development platform [7], providing a syntax and a type checker, a proof obligation generator, an interactive theorem prover, as well as an animator. The generation and the validation of proof obligations are concerned with the static properties of a specification. On the other hand, the animation of a specification explores its state set with the help of model checking techniques. The animator searches for valid assignments to the model instance variables, and checks whether the explored model states meet the defined invariants.

Reusability is enabled by modularized TCB specifications. One approach is to establish an inheritance hierarchy of specifications, which allows to derive the specifications of security models and their instances from the specification of the common model core. However, inheritance is yet not supported by existing specification languages and their tools. For this reason, we follow a different approach based on an inclusion hierarchy of *contexts*. A context defines the elements of a specification by a set of *base types* T_i, a set of typed *constants* c_i, a set of *axioms* $\mathsf{A}_i(c)$ defined on the constants, and a set of *theorems* $\mathsf{B}_i(c)$. The latter are derived from the axioms for their validation. A specification of a security model is contained by such a context and each model instance specification then includes the model context to specify its state machine. The advantage of this approach is that it is supported by Event-B via *extensions*: a context can be extended by other contexts, which results in the unification of all extended contexts. Figure 5.2 shows a context hierarchy in Event-B. A model's specification may consist of several contexts model_i, depending on its complexity. Each context may extend a context model_common that contains shared specification components. The complete specification of a model in model_context then unifies all involved context, and can be used in the instance_context by any model instance. The

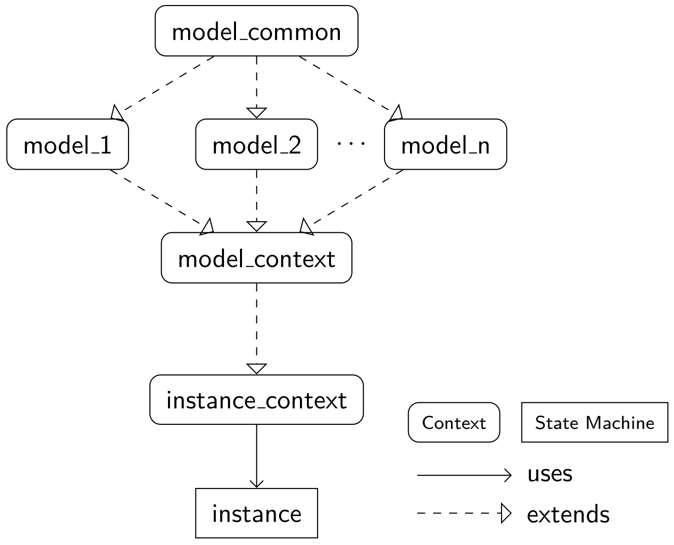

Figure 5.2: Context Hierarchy in Event-B [178]

instance_context specifies the individual properties of a model instance, on which the state machine of the model instance (instance) is defined.

Following this approach, TCB specifications are based on a coarse-grained modularization, applying the concepts of existing specification languages and tools. Future work may deal with developing a domain-specific specification language that provides inheritance concepts; see Section 6.5 for a detailed discussion of the specification results.

The basis of a TCB specification with a *small size and low complexity* is a precise mathematical notation, which allows for controlling specification redundancy while stepwise rewriting a model instance and its TCB functions. In case redundancy is required for error avoidance, we document it and proof its equivalence via theorems. Moreover, in a modularized specification such as shown in Figure 5.2, the specification complexity should be contained in the model contexts. Specifying a model instance, which is a constantly repeating process, then has lower complexity and thus is less error-prone.

5.4 TCB Specification Method

The goal of this section is to present a specification method based on the approaches of Section 5.3, whose output are formal specifications that can serve as basis for automated code generation of a causal TCB's policy-dependent RTE. The approach is to develop a general specification method for sequential, state-based systems that can be configured for a variety of state-based specifications languages. This section discusses each step of the specification method in an abstract way and demonstrates, how each step can be configured for Event-B specifications. A comprehensive discussion of the abstract specification method, the special configuration cases for Event-B, and necessary proof obligations is provided in [178].

The specification method has to rewrite all model and model instance components within a set of contexts. Having rewritten these components and proofed that all necessary theorems are met, it remains to refine the specification by the derived ADTs and their operators. Here, the goal is to dissolve the mathematical specification of the state set and the extension vector to a more implementation-oriented one by using the ADTs that are derived by causal dependencies. In doing so, we apply specific refinement techniques (see [178]) to proof the correctness and consistency of the resulting formal specification regarding the underlying model instance. As a result, the generated code is similar to the derived TCB functions in terms of the inheritance hierarchy, types, and functional perimeter (Section 4.4.4). However, due to the validated formal specification and automated code generation, we provide the basis for validating the correctness and consistency of the resulting program code with respect to the model instance.

In the following, we briefly present the formal specification a security model and a corresponding model instance. While Section 5.4.1 is concerned with rewriting the model components, Section 5.4.2 rewrites the components of the model instance.

5.4.1 Specification of a Core-based Security Model

To formally specify a core-based security model, the following model components have to be contained in a TCB specification:

- Codomains of the primitive elements, e.g., roles, objects, or users,
- Extension vector,
- State set,
- Primitive operations, and
- Conditions.

This section stepwise rewrites each of these model components first in an abstract way and afterwards in Event-B notation.

Codomains of Primitive Elements The codomains of the primitive elements of core-based security models are pairwise disjoint sets, which can be represented by base sets in a formal specification. Codomains and base sets are semantically similar; thus, we merely have to change the notation.

In Event-B we can define base sets in the `SETS` clause of a context. The base sets $\mathtt{SET_1}, \mathtt{SET_2}, \ldots, \mathtt{SET_n}$ are introduced in the context `model_common` as follows (Figure 5.2).

```
CONTEXT model_common
SETS
  SET1
  SET2
  ...
  SETn
END
```

Extension Vector The extension vector $E = (e_1, e_2, ..., e_h), h \geqslant 0$, contains a set of static model elements, whose values are application-specific and do not change during policy runtime. Each element can have specific properties, e.g., transitivity or symmetry of relations, and elements can be combined by statements in predicate logic.

Since the context of a specification contains only constants (instead of variables) and we have argued for specifying a model in a hierarchy of contexts (Figure 5.2), all extension vector elements must be expressed by constants. We refrain from encapsulating the constants within a tuple, since the latter unnecessarily increases the complexity without having any advantages. That means, for each e_i we add a constant c to the specification. The properties of each element are expressed as axioms A(c). Moreover, for complex properties we also add alternative expressions as theorems B(c), since this reduces not only possible errors but also the effort for proofing the axioms.

In Event-B constants are specified by the clause `CONSTANTS` and axioms are introduced by `AXIOMS`. For each e_i the specification thus defines a constant $\texttt{stat}_\texttt{i} \in \texttt{STAT}_\texttt{i}$ in the context `model_common`. Axioms that express the elements' properties have to be defined in connection with the constants' typing axioms.

```
CONTEXT model_common
SETS
  ...
CONSTANTS
  stat_1     > names
  stat_2
  ...
  stat_n
AXIOMS
  st1_t: stat_1 ∈ STAT_1     > typing
  st2_t: stat_2 ∈ STAT_2
  ...
  stn_t: stat_n ∈ STAT_n
END
```

State Set The model's state set $Q = \times_i^n D_i, n \geqslant 1$, must also be represented in a formal specification as a constant STATE of the same type. The specification of a model instance then uses a single state variable state ∈ STATE. Primitive operations and conditions are also specified based on this constant. For example, consider prim_i as a specification of the primitive operation $prim_i$, its type is defined as

$$\mathsf{prim}_i : \mathsf{STATE} \times \overbrace{\quad\cdots\quad}^{\text{more parameters}} \rightarrow \mathsf{STATE}.$$

To ensure a specification's inner consistency, we have to define consistency conditions based on STATE. To this end, we define a set of conditions and a subset CONSISTENT_STATE ⊆ STATE that only contains consistent states. The specification of a model instance then has to ensure that state ∈ CONSISTENT_STATE. However, primitive operations must still be defined on STATE. That means, for each command of the authorization scheme we have to validate that the result of all sequentially executed primitive operations is in the set of consistent states.

In Event-B STATE is defined in the context model_common by the state components $\mathtt{dyn_i\_q} \in \mathtt{DYN_i}$. For each state component we additionally define a projection function $\mathtt{dyn_1\_q}, \mathtt{dyn_2\_q}, \ldots, \mathtt{dyn_n\_q}$ that, based on the Event-B operations prj1 and prj2, selects the first or second element of a tuple. Thereby, tuples of type $x_1 \mapsto x_2 \mapsto \cdots \mapsto x_n$ are dissolved beginning at the last element, i.e., the result of prj1 for this tuple is $x_1 \mapsto x_2 \mapsto \cdots \mapsto x_{n-1}$. The definition of all projection functions is comprised by the axiom prj_d.

Consistency conditions are defined on top of the state set. For example, the conditions $\chi_1, \chi_2, \ldots, \chi_m$, each of which depending on at least one state component $\mathtt{dyn_1\_q}, \mathtt{dyn_2\_q}, \ldots, \mathtt{dyn_n\_q}$, are defined as part of STATE. Additionally, we exclude interim states by requiring that state ∈ CONSISTENT_STATE and conditions $\chi'_1, \chi'_2, \ldots, \chi'_k$. We mark the property cst_1 as theorem such that it must be validated.

```
CONTEXT model_common
CONSTANTS
  STATE
  dyn₁_q
  dyn₂_q
  ...
  dynₙ_q
AXIOMS
  st: STATE = {dyn₁ ↦ dyn₂ ↦ ··· ↦ dynₙ |
          dyn₁ ∈ DYN₁ ∧ ··· ∧ dynₙ ∈ DYNₙ ∧
          χ'₁ ∧ χ'₂ ∧ ··· ∧ χ'ₖ }
  cst: CONSISTENT_STATE = {dyn₁ ↦ dyn₂ ↦ ··· ↦ dynₙ |
          dyn₁ ↦ dyn₂ ↦ ··· ↦ dynₙ ∈ STATE ∧
          χ'₁ ∧ χ'₂ ∧ ··· ∧ χ'ₖ }
  cst_1: CONSISTENT_STATE ⊆ STATE     theorem
  dyn₁_t: dyn₁_q ∈ STATE → DYN₁
  ...
  dynₙ_t: dynₙ_q ∈ STATE → DYNₙ
  prj_d: ∀q · q ∈ STATE ⇒
          dyn₁_q = prj1(prj1(prj1(...(prj1(q))))) ∧
          dyn₂_q = prj2(prj1(prj1(...(prj1(q))))) ∧
          ...
          dynₙ_q = prj2(q)
END
```

Primitive Operations Each command of the authorization scheme of a model instance uses primitive operations to model state transitions. Thus, the specification of the model instance must be able to access the model's primitive operations. As a consequence, for each $prim_i \in PRIM$ we specify a constant prim_i of type $\mathsf{prim}_i : \mathsf{STATE} \times \mathsf{T}_1^{prim_i} \times \mathsf{T}_2^{prim_i} \times \cdots \times \mathsf{T}_n^{prim_i} \rightarrow \mathsf{STATE}$, where $\mathsf{T}_j^{prim_i}$ represent the input parameter types of the primitive operation that have replaced the input vector X during algebra engineering (Section 4.4.2.1). For example, the operation $addUsers : State \times Set[User] \rightarrow State$ is rewritten as constant of type

$$\mathsf{addUsers} : \mathsf{STATE} \times \mathcal{P}(\mathsf{USER}) \rightarrow \mathsf{STATE}.$$

Almost any primitive operation has specific properties (idempotence, reversibility, or commutativity), which we define as additional theorems $\mathsf{B}(\mathsf{c})$. This contributes to a specification's correctness, since violations of these properties lead to unprovable theorems.

In Event-B each $prim_i$ is specified as constant within an individual context `model_i` (Figure 5.2). Besides the definition of an operation's name and type, a specification contains its semantics by a pre- and a postcondition. The postcondition is specified by defining the modifications of q in tuple notation. That means, in the following Event-B scheme the expression q' has to be replaced by a specific state tuple. The precondition is defined ahead within round brackets. To ensure inner consistency of the specification, we define an additional theorem $\mathtt{prim_i\_\_v}$ for each primitive operation, which validates that the subsequent state $\mathsf{q}' \in \mathtt{STATE}$.

```
CONTEXT model_i
EXTENDS
  model_common
CONSTANTS
  prim_i
AXIOMS
  prim_i_t: $\mathtt{prim_i} \in \mathtt{STATE} \times \mathtt{T}_1^{\mathtt{prim_i}} \times \mathtt{T}_2^{\mathtt{prim_i}} \times \cdots \times \mathtt{T}_n^{\mathtt{prim_i}} \rightarrow \mathtt{STATE}$
  prim_i_d: $\forall \mathtt{q}, \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \cdot ($
            $\mathtt{q} \in \mathtt{STATE} \land$
            $\mathtt{a}_1 \in \mathtt{T}_1^{\mathtt{prim_i}} \land \mathtt{a}_2 \in \mathtt{T}_2^{\mathtt{prim_i}} \land \cdots \land \mathtt{a}_n \in \mathtt{T}_n^{\mathtt{prim_i}})$
        $\Rightarrow \mathtt{prim_i}(\mathtt{q} \mapsto \mathtt{a}_1 \mapsto \mathtt{a}_2 \mapsto \cdots \mapsto \mathtt{a}_n) = \mathtt{q}'$
  prim_i_v: $\forall \mathtt{q}, \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \cdot ($
            $\mathtt{q} \in \mathtt{STATE} \land$
            $\mathtt{a}_1 \in \mathtt{T}_1^{\mathtt{prim_i}} \land \mathtt{a}_2 \in \mathtt{T}_2^{\mathtt{prim_i}} \land \cdots \land \mathtt{a}_n \in \mathtt{T}_n^{\mathtt{prim_i}})$
        $\Rightarrow \mathtt{q}' \in \mathtt{STATE}$
END
```

For example, the post- and the precondition of the commando `addUsers()` are defined as follows. While the postcondition (1. statement) defines the state transition by adding a set of new users to the user set contained in the state, the precondition (2. statement) restricts the execution of `addUsers()` to those input sets that do not contain already existing users.

$$\mathtt{addUsers}(\mathtt{q} \mapsto \mathtt{u}) = (\mathtt{U\_q}(\mathtt{q}) \cup \mathtt{u}) \mapsto \mathtt{S\_q}(\mathtt{q}) \mapsto \mathtt{UA\_q}(\mathtt{q}) \mapsto \mathtt{user\_q}(\mathtt{q}) \mapsto \mathtt{roles\_q}(\mathtt{q}))$$

$$\forall \mathtt{q}, \mathtt{u} \cdot (\mathtt{q} \in \mathtt{STATE} \land \mathtt{u} \in \mathbb{P}(\mathtt{USER}) \land \mathtt{u} \cap \mathtt{U\_q}(\mathtt{q}) = \varnothing) \Rightarrow \mathtt{addUsers}(\mathtt{q} \mapsto \mathtt{u}) = \ldots$$

Conditions Conditions guard the execution of primitive operations within the commands of an authorization scheme. That means, they also have to be accessible by the specification of a model instance. We hence apply the same specification method: for each $cond \in COND$ we add a constant cond_i of type $\mathsf{cond}_i : \mathsf{STATE} \times \mathsf{T}_1^{cond_i} \times \mathsf{T}_2^{cond_i} \times \cdots \times \mathsf{T}_m^{cond_i} \rightarrow \mathsf{BOOL}$, where $\mathsf{T}_j^{cond_i}$ are the individual input types of the replaced input vector X and BOOL is the base set of Boolean values.

In Event-B specifying conditions is less complex than specifying primitive operations, since they map to Boolean values instead of the state set such that we do not have to define axioms for inner consistency. In analogy to primitive operations, we add a constant `cond_i__t` for each $cond_i$. We use the function `bool(f)` to represent the Boolean value of a logic statement `f`. That means for a specific condition, $\mathtt{f}(\mathtt{q}, \mathtt{a}_1, \ldots, \mathtt{a}_\mathtt{n})$ is to be replaced by the statement that expresses the Boolean value of the condition. All conditions are subsumed in the context `model__cond`.

```
CONTEXT model_cond
EXTENDS
  model_common
CONSTANTS
  cond_i
AXIOMS
  cond_i_t: $\mathtt{cond_i} \in \mathtt{STATE} \times \mathtt{T_1^{cond_i}} \times \mathtt{T_2^{cond_i}} \times \cdots \times \mathtt{T_m^{cond_i}} \rightarrow \mathtt{STATE}$
  cond_i_d: $\forall \mathtt{q}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_m} \cdot ($
            $\mathtt{q} \in \mathtt{STATE} \wedge$
            $\mathtt{a_1} \in \mathtt{T_1^{cond_i}} \wedge \mathtt{a_2} \in \mathtt{T_2^{cond_i}} \wedge \cdots \wedge \mathtt{a_m} \in \mathtt{T_m^{cond_i}}$
        $) \Rightarrow \mathtt{cond_i}(\mathtt{q} \mapsto \mathtt{a_1} \mapsto \mathtt{a_2} \mapsto \cdots \mapsto \mathtt{a_m}) = \mathtt{bool}(\mathtt{f}(\mathtt{q}, \mathtt{a_1}, \ldots, \mathtt{a_m}))$
END
```

At this stage of specification engineering all components of a core-based model are rewritten. It now remains to transform the individual properties of a specific model instance of this model in a formal specification.

5.4.2 Specification of a Model Instance

Specifying a model instance is concerned with rewriting all components of the model instance in the notation of a formal specification, thereby reusing the model specification of Section 5.4.1:

- Elements of the codomains,
- Initialization of the extension vector,
- Initial state, and
- Authorization scheme.

This section rewrites each model instance component in an abstract way and in Event-B notation.

Elements of the Codomains A model instance names the specific elements of a model's codomains. Thus, we have to specify each element of the previously defined base sets. This is done by specifying a constant $\mathtt{e}_j \in \mathsf{T}_i$ for each primitive element $e_j \in T_i$, where T_i is the specification of codomain T_i. On top of that, we have do specify axioms, which ensure that all $\mathtt{e}_j$ are pairwise disjoint. In case all elements of a codomain are specified by a model instance, an additional axiom $\mathsf{T}_i = \{\mathtt{e}_{j(1)}, \mathtt{e}_{j(2)}, \ldots, \mathtt{e}_{j(n)}\}$ is required that defines the completeness of the base set T_i.

In Event-B elements $\mathtt{e_1}, \ldots, \mathtt{e_n}$ of the base sets $\mathtt{SET_i}$ are specified as constants, and they are mapped to their base sets via axioms $\mathtt{SET_i\_p}$. The function $\mathtt{partition}(\mathtt{M}, \mathtt{P_1}, \mathtt{P_2}, \ldots)$ defines $\mathtt{P} = \{\mathtt{P_1}, \mathtt{P_2}, \ldots\}$ as a partition of $\mathtt{M}$. By mapping each $\mathtt{e_j}$ to another subset of the partition, we can express that they are pairwise disjoint. On top of that, $\mathtt{partition}()$ also types $\mathtt{e_j}$ by mapping it to its base set $\mathtt{SET_i}$.

```
CONTEXT instance_context
EXTENDS
  model
CONSTANTS
  e1
  e2
  ...
  en
AXIOMS
  SETi_p: partition(SETi, {e1}, ..., {en})
END
```

Initialization of the Extension Vector A model instance also defines the values of the extension vector elements. In a specification we have to map these values to the vector elements by defining axioms in the model instance context. These axioms have to be consistent with respect to the axioms that define the elements' types and properties. That means, for each extension vector element e_i with value v_i and specification stat_i, we have to define an axiom $\mathsf{stat}_i = v_i$.

In Event-B this is done in within `instance_context` with axioms $\mathtt{stat_i} : \mathtt{stat_i} = \mathtt{v_i}$.

Initial State Before specializing the initial state of the model instance, we have to define the state set of the model instance specification. This is done by specifying a variable state as element of STATE or $\mathsf{CONSISTENT\_STATE}$ and invariants $\mathsf{I}(\mathsf{x}, \mathsf{c})$. The initial state $\mathsf{v}(\mathsf{c})$ then defines specific values $q_0 = (d_{1_0}, ..., d_{n_0})$, where $d_i \in D_i$; the definition must be correct with respect to state and the invariants $\mathsf{I}(\mathsf{x}, \mathsf{c})$.

In Event-B the state set is defined as part of the state machine specification by using the clauses `VARIABLES` and `INVARIANTS`. The state machine of a model instance is specified as a `MACHINE` called `instance`. By means of axiom `inv1`, it defines the instance's `state` as element of `CONSISTENT_STATE` , which has already been specified in `model_common`. To ease the validation of theorems, an additional theorem `inv2` specifies $\mathtt{state} \in \mathtt{STATE}$, which is `true` since $\mathtt{CONSISTENT\_STATE} \subseteq \mathtt{STATE}$. The state variable is initialized by the Event-B operation INITIALISATION, which maps values to a given variable. Here, we have to use the tuple notation, because the state is defined as Cartesian product.

```
MACHINE instance
SEES instance_context
VARIABLES
  state
INVARIANTS
  inv1: state ∈ CONSISTENT_STATE
  inv2: state ∈ STATE    theorem
END
EVENTS
  INITIALISATION ≙
    begin
      act1: state := d1 ↦ d2 ↦ ··· ↦ dn
    end
END
```

Authorization Scheme The authorization scheme contains a set of state-modifying and non-state-modifying commands, which define the dynamic behavior of a model instance via the state transition function and the output function. It now remains to add the authorization scheme to the specification of the model instance, in order to specify the behavior of the state-based system. In doing so, we define each command of the authorization scheme as operation $\mathsf{o}_i : [\mathsf{p}_i, \mathsf{pre}_i, \mathsf{post}_i]$ of the state machine: the parameters p_i of a command are contained in the codomains T_i of a model, which have already been specified by the base sets of the model specification. A command's conditions are subsumed by the operation's precondition pre_i; each condition of the model is already specified by cond_i. In analogy, a command's primitive operations are subsumed by the operation's postcondition post_j, where each primitive operation is already specified by prim_i. The postcondition is defined as $\mathsf{x}_k := \mathsf{a}(\mathsf{p}_i, \mathsf{x}, \mathsf{c})$, where x_k is a state variable and a is a value that is calculated in dependence on the parameters, the predecessor state, and some constant, by sequentially executing a command's primitive operations. For example, the command *login*() of the RBAC HIS policy (Appendix B.2) is specified as

$$\mathsf{state} := \mathsf{mapUserSessions}(\,\mathsf{createSessions}(\mathsf{state}, \{\mathsf{s}\}),\ \{(\mathsf{s}, \mathsf{u})\}),$$

where $\mathsf{mapUserSessions}, \mathsf{createSessions}, \mathsf{s}$, and u are the specified model components with the same names. The only difference of non-state-modifying operations is that they return a value of type BOOL instead of STATE.

In Event-B the operations are specified by EVENTS. Preconditions are called *guards*, and an event can only occur if all guards (statements in predicate logic) are met. Postconditions are specified as assignments $\mathtt{x} := \mathsf{a}$. From this it follows, that for each command of the authorization scheme we have to define an event and rewrite its conditions as guards, using the previously defined $\mathtt{pcond_i}$. A command's consequent is rewritten as an assignment, which assigns a new value (computed by the command's primitive operations) to the `state` variable.

```
MACHINE instance
...
EVENTS
  event_i ≙
    any
      p_1      ▷ parameters
      ...
      p_n
    where
      grd1: G_1      ▷ guards
      ...
      grdm: G_m
    then
      act: state := a      ▷ primitive operations
    end
END
```

To ensure the inner consistency of the specification, axioms have to be defined that prove that the resulting states are contained in the specification's state set. The successful validation of these axioms then guarantees that based on a consistent initial state and by applying the machine's events under consideration of their guards, only consistent successor states can be reached. This is a significant contribution to the specification's correctness and inner consistency.

5.5 Summary

This section has presented a general specification method for arbitrary security models in core notation along with their derived TCB function sets. Even though we have configured this method for Event-B due to is language features and sophisticated tool support, it can easily be configured for other specification languages for sequential, state-based systems.

We have applied the specification method for Event-B to the RBAC HIS policy. Appendix D contains the complete TCB specification; further refinements are illustrated in [178]. First, we have specified a set of contexts for the core-based RBAC model of Section 3.4.5. By extending the unified model context, we then have specified the state machine for the RBAC HIS model instance presented in Appendix B. Thereby, we have demonstrated that the specification method can be applied to a core-based model instance that is inspired by a real-world RBAC policy. The results of doing so are discussed in Section 6.

6 Evaluation

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

This dissertation has developed a method for systematically engineering TCBs based on causal dependencies between security policies and TCB functions. The foundation of causal TCB engineering are security models in core notation to formally express the policies that are present in causal TCBs. A TCB's functional perimeter is then derived from core-based models by a rule system that formalizes said causal dependencies.

To the best of our knowledge, this is the first approach that identifies a TCB's components already at the functional level. Many approaches such as [123, 143, 169–171, 227] also aim at reducing the size of a system's TCB; however, they tackle the problem at the implementation level.

The precise identification of a TCB's functional perimeter leads to a set of merits that are hard to achieve at the level of implementation. The main merits are: (i) it allows for implementing a TCB in a safe environment that indeed can be isolated from untrusted system components. (ii) It defines the scope of system verification. That means, by identifying a TCB's functions and by strictly isolating them from untrusted system functions, these and only these functions have to be verified at implementation level.

The goal of this chapter is to evaluate whether causal TCB engineering indeed is able to precisely identify a TCB's functional perimeter. To this end, we evaluate if the functional perimeter is as small as possible with respect to a TCB's policies. Additionally, this section aims at evaluating the feasibility of the model-based engineering approach for real-world application scenarios. Here, it is significant to show for instance that the functions of a causal TCB can be straightforwardly implemented with low effort and that the implementation process is supported by adequate methods and tools.

The application scenarios for causal TCB engineering are manifold. The ones that mostly benefits are: policy-controlled OS, embedded systems, and policy engineering. Policy-controlled operating systems benefit in that a TCB's functional perimeter exactly defines the OS components that have to be trusted and hence isolated. This contributes to setting the course for formal verifications of operating systems and hence for guaranteeing their security properties. In the context of embedded systems, causal TCBs help to identify the number of required trusted functions. This is important since an increasing amount of code that even might not be needed for the limited functionality of embedded systems increases the memory footprint, which is considered an important constraint for embedded systems [66]. On the other hand, causal TCB engineering indirectly supports policy engineers via the common model core. By using models in core notation, engineers can exploit model-independent methods and tools for model engineering and model analysis, which improves not only the efficiency but also the effectiveness of policy engineering.

On this account, Section 6.1 introduces four evaluation goals and motivates the chosen evaluation methods. The following sections then accomplish the evaluation by applying the chosen methods. Since this dissertation focuses on methods to identify and implement the functions of a TCB's policy-dependent RTE, the evaluation goals are directed to the engineering methods of the policy-dependent RTE. Evaluating the policy-independent RTE, particularly implementing it on a specific implementation platform, remains future work (Section 8).

6.1 Evaluation Goals and Methods

The main evaluation goal is to show that causal TCB engineering is able to precisely identify the functional perimeter of TCBs. This requires that the functional perimeter derived from a TCB's policies is as small as possible with respect to the present security policies and thus may not contain *functional redundancy.* In contrast to specification engineering, where dedicated specification redundancy results in a set of advantages as discussed in Section 5.2, redundancy of TCB functions only increases the functional perimeter and hence has to be avoided.

The functional design of causal TCBs already deals with functional redundancy when more than one policy is present. In this case, all model functions that are shared by the policies are subsumed in an additional sub-layer to avoid functional redundancy across different sets of model functions (Section 4.1.2). However, redundancy can still occur within the derived model function sets if engineering a TCB's policy-dependent RTE does not consider that. For this reason, we evaluate how the derivation of the TCB functions deals with functional redundancies in Section 6.2. Here, the evaluation method is to qualitatively argue how ADT engineering avoids functional redundancy.

To evaluate the feasibility of the model-based approach of causal TCB engineering for real-world application scenarios, we discuss three additional evaluation goals that are inspired by the applied methods of this dissertation:

- Expressive power and modeling effort of core-based model engineering,
- Implementation effort for policy substitution, and
- Formal TCB specifications as basis for implementing the policy-dependent RTE.

In order that causal TCBs can be applied in real-world systems, they must be able to support a wide variety of significant security policies. This in turn means, that the model core must be able to express a wide range of security models, since policies are formalized by core-based models to derive a TCB's functional perimeter. Consequently, we have to evaluate the *expressive power of the model core and core-based model engineering* with respect to the models they can adequately express. On the other hand, the *effort for engineering a model in core notation* must not be higher than compared to engineering arbitrary models without the model core. Only then is core-based model engineering attractive to be applied to formalize real-world policies.

The applied evaluation method is based on demonstrating the model types the model core can adequately express. This we have already done in detail in Sections 3.4 and 3.5.1. Due to completeness reasons, Section 6.3 briefly revisits the results of these discussions. On top of that, it summarizes the involved modeling effort.

Supporting the substitutionality of policies is a significant property of causal TCBs. The substitution of policies generally is supported by the functional TCB design in that the policy-dependent RTE of other policies is not affected when modifying the functional perimeter of one policy. It now remains to evaluate if the same statement can be applied in the context of a TCB's implementation. Here, we must analyze the *implementation effort for the policy-dependent RTE*, whenever a policy requires to modify a TCB's functional perimeter. Only in case this is in fact as low as conceptually designed, the implementation of causal TCBs in real-world applications is feasible.

To evaluate the implementation effort (Section 6.4), we first build a prototype implementation of a policy-dependent RTE that is tailored to enforce the RBAC HIS policy introduced in Section 3.4.4. The prototype is implemented as a *C++* user mode program that implements the derived TCB functions of Section 4.5. By this means, we first show that functions derived by ADT engineering can be straightforwardly implemented. Additionally, we analyze the implementation effort for loading and unloading TCB functions based on this prototype. Here, we distinguish between different cases: (i) adding a novel policy, (ii) removing a policy, and (iii) modification of a policy.

Causal TCBs are developed individually for each application scenario. In the context of TCB implementation, this leads to that the gap between TCB functions described by ADTs and a platform-specific TCB implementation is bridged every time anew. To provide method and tool support for this step, we have developed a specification method for *formal TCB specifications* in Section 5.4. It remains to evaluate if these TCB specifications produced by the specification method can serve as basis for implementing a TCB's policy-dependent RTE for real-world scenarios. That means, we evaluate in Section 6.5 if the resulting TCB specifications meet the requirements that we have identified to be fundamental for a TCB specification to serve as basis for an implementation. The evaluation method is to qualitatively argue how TCB specifications meet said requirements. This argumentation is supported by the example TCB specification for the RBAC HIS policy.

6.2 Functional Redundancy of the Policy-dependent RTE

While Section 4.1.2 has discussed how the functional TCB design deals with functional redundancy, the goal of this section is to show that engineering a TCB's policy-dependent RTE results in a nonredundant functional perimeter. As introduced in Section 4.4.1, engineering a policy-dependent RTE consists of two steps: algebra and ADT engineering. While algebra engineering rewrites a core-based model in a set of algebras, ADT engineering derives the functions of the policy-dependent RTE from these algebras. Here, two kinds of functional redundancy may occur: functional redundancy that crosses the borders of multiple `security model functions` components (Section 6.2.1) and functional redundancy within a single `security model functions` component (Section 6.2.2). The following sections qualitatively discuss the engineering principles applied by ADT engineering that aim at a nonredundant functional perimeter of the policy-dependent RTE.

6.2.1 Redundancy across Multiple `Security Model Functions` Components

ADT engineering derives the functional perimeter of the policy-dependent RTE from core-based models instead of model instances. That means, for each even slightly different security model, a separate `security model functions` component is integrated in the policy-dependent RTE. Depending on the degree of model similarities, functional redundancy across the multiple `security model functions` components may thus occur.

To avoid this kind of functional redundancy, ADT engineering applies the notion of subsuming security models (Section 4.4.2). By unifying primitive actions and clauses of models with shared state spaces and extension vectors, the subsuming model constitutes an upper bound for the functional perimeter of the required policy-dependent RTE. ADT engineering then derives exactly one `security model function` component for each subsuming model, containing all functions that are required by the subsumed models. By this means, functional redundancy across multiple `security model functions` components can no longer occur.

6.2.2 Redundancy within a `Security Model Functions` Component

Based on a set of component algebras for each security model, ADT engineering generates a set of ADT operators, which represent the TCB functions that are needed to implement the algebras. In doing so, ADT engineering produces a separate operator set for each component algebra and afterwards unifies all derived sets. The resulting set contains all functions that are required to implement the underlying model. In case a TCB enforces multiple models, this procedure must be applied to each model. The last step is then to unify all model function sets such that the resulting set contains all functions of a TCB's policy-dependent RTE.

In general, unifying TCB function sets follows the semantics of the set union operator, whose basic property is to avoid redundant set elements. That means, we have mapped the problem of a redundancy-free TCB function set to identifying whether a function is a member of more than one function set. This we have solved as follows.

Each TCB function is described by a name, a domain and a codomain, an access status, and an optional return value (Definition 4.3). The semantics of a function is specified either by the mapping $\beta : \mathscr{P}(\mathcal{O}) \to \mathcal{U}_O$, which maps a set of ADT operators to a universe operation, or by axioms of the ADT that the function belongs to. However, in both cases it is true that functions with the same semantics have the same name. Thus, a name can be considered a unique identifier for a TCB function, independent of any other function attributes. As a consequence, the set union of TCB function sets avoids redundancy by simply comparing the functions' names.

6.2.3 Summary

ADT engineering has to deal with two kinds of functional redundancy: across multiple `security model functions` components and within a single `security model functions` component. The first kind of redundancy is avoided by applying the notion of subsuming security models to constitute an upper bound for the functional perimeter of a policy-dependent RTE. On the other hand, functional redundancy within a `security model functions` component is avoided by a set union of the derived TCB function sets.

From this it follows that ADT engineering produces TCB function sets that are as small as possible with respect to a TCB's policies, since they do not contain functional redundancy. As a consequence, the functional perimeter of causal TCBs is exactly defined.

On top of that, the algebraic approach, the formal rule system, as well as the set-theoretic approach to avoid functional redundancy are the key enablers for an automated tool that composes the functional perimeter of a causal TCB. Output of such a composition tool then is a set of TCB functions that serves as basis for a TCB's implementation. This contributes to unlocking causal TCB engineering to manifold application scenarios.

6.3 Expressive Power and Modeling Effort of Core-based Model Engineering

To evaluate the feasibility of the model-based approach of causal TCB engineering for real-world application scenarios, this section briefly revisits the results of core-based model engineering with respect to its expressive power and the involved modeling effort. Section 3.5.1 has argued that the computational power of the model core is precisely equivalent to a Turing Machine [107, 108, 233, 234]. To reason for the core's express power to be applicable for real-world policies, however, analyzing the computational power alone is not sufficient. We also have to consider its ability to adequately formalize security models.

First, we have identified the common model core in Section 3.1 by a security model family tree that has revealed that deterministic automata already are immanent parts of existing models like [26, 35, 69, 92, 101, 106, 204, 209, 254]. In the following, we have generalized the model-specific automata in such a way that these models have become specific instances of the common model core.

We have demonstrated in detail in Sections 3.3 and 3.4 that the model core adequately expresses the HRU model [107, 108], Sandhu's RBAC$_3$ model [212] and hence all models of Sandhu's RBAC model family, an MLS model for web services [119, 120] that is based on the well-known BLP model [26, 27], and a general ABAC model that can be specialized to the needs of specific application scenarios that use attributes for access control decisions [251]. From this it follows that core-based model engineering can be applied to (i) IBAC, RBAC, and ABAC models, each of which in their DAC and MAC variant, and (ii) information flow models that use lattices for modeling information flow rules. On top of that, we have shown in Section 3.5.1 that core-based model engineering can also adequately express trust management models by means of the RT model family of [159, 160].

The basis of the expressive power of core-based model engineering is the model core that is minimal with respect to domain-specific model abstractions. All it remains to do is to specialize the core's components, which significantly eases the burdensome process of model engineering. In the case of re-engineering an existing model, the modeling effort depends on the model to be re-engineered. If it already supports policy dynamics by means of a deterministic automaton, the effort is limited to rewriting the model components in core notation. If it is static, the modeling effort depends on the modeling goal: the modeling effort to rewrite a static model in a static core-based model is limited to specializing the core's extension vector E. In case the static model is to be enhanced by policy dynamics in core notation, the state set Q, the input set Σ, and the state transition function δ also need to be specialized.

In summary, we have provided proof that the common model core along with core-based model engineering can adequately express a wide variety of security models that is significant for modeling real-world security policies. The basis of the model core's expressive power is its minimality with respect to domain-specific model abstractions. A policy engineer only has to specialize the deterministic automaton along with the static extension vector. Therefore, core-based model engineering considerably contributes to unlocking causal TCB engineering for real-world applications scenarios.

6.4 Implementation Effort for Policy Substitution

The goal of this section is to compare the effort for modifying the functional perimeter of a policy-dependent RTE, which we have discussed based on the functional design of a causal TCB in Section 4.2, to the actual effort involved in a TCB's implementation. To this end, we have developed a prototype implementation of a policy-dependent RTE to serve as basis for analyzing the implementation effort. Section 6.4.1 gives a short overview of this prototype implementation. The implementation effort that is involved when policies are loaded, respectively unloaded, or when the functional perimeter of a present policy is modified, is then discussed in Section 6.4.2. Note that even though this discussion is based on a platform-independent user mode program, all statements are also true for an analogous platform-dependent implementation within a policy-controlled OS.

6.4.1 Prototype Implementation of a Policy-dependent RTE

The prototype implements a policy-dependent RTE, which is tailored to the RBAC HIS policy introduced in Section 3.4.4. The basis for the prototype implementation are the TCB functions that we have derived from the core-based RBAC HIS model in the process of ADT engineering in Section 4.5.1.

The naïve approach for implementing the derived TCB functions is to generate the program code based on the class diagram illustrated in Figure 4.28. However, this approach thwarts the functional design of causal TCBs (Figure 4.3), which distinguishes not only between security model functions and model instance functions, but also between shared model functions and individual model functions to avoid functional redundancy. To follow this approach in a TCB's implementation, we thus have to determine first, which of the derived TCB functions are specific for the RBAC HIS model and which may be shared by other models. We subsequently discuss a model-independent rule for doing so.

As discussed in Section 6.2.1, it can never happen that two `security model functions` components within a policy-dependent RTE contain the same functions to implement the models' state set and extension vector. Hence, these functions are specific for each `security model functions` component. In contrast, it may happen that multiple model implementations use (some of) the same primitive types (e.g., roles or users). However, these models usually belong to the same model domain and since we aim at collecting model functions that are shared beyond model domains, primitive types are not considered shared model functions. From this it follows that the functions to implement a model's primitive types, state set, and extension vector are model-specific. On the contrary, the remaining functions are shared model functions. The reason is that these functions are derived from the algebras of the unified algebra foundation (Section 4.4.2.1), which can express a wide variety of core-based models in algebra notation. Thus, deriving TCB functions from these shared algebras naturally results in shared model functions.

Based on this rule we have structured the classes that implement the policy-dependent RTE for the RBAC HIS policy as illustrated in Figure 6.1. For each ADT we have generated a class with the same name, and we have added this class either to the package *rbac-his-model*, representing a specific `security model functions` component, or to the package *shared-model-functions*. Note that we thereby have retained the individual types of the model's basic elements rather than collapsing them into a single one that may be implemented either by a platform-specific primitive type such as *int* or *string*, or by a single object type. As a result, all derived TCB functions are contained in either one of these two packages. The next step has been to implement the functions and compose the program code of the `executable RBAC HIS policy`.

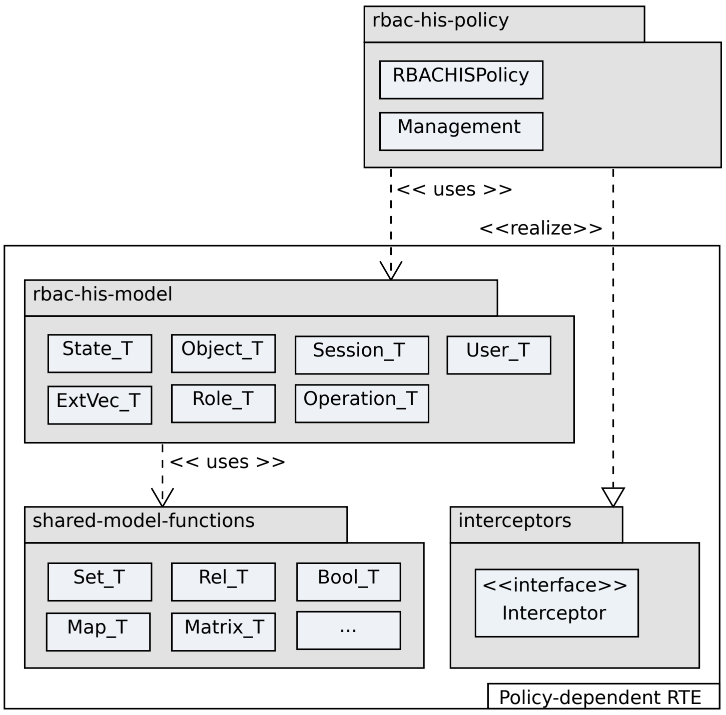

Figure 6.1: Prototype Implementation of the Policy-dependent RTE

All classes that implement the `executable RBAC HIS policy` are contained in the package *rbac-his-policy*. The class *RBACHISPolicy* basically contains two member variables representing the policy's current state and extension vector, on which the authorization scheme is implemented as shown in Figure 4.28. A policy's management strategies, e.g., for thread or memory management, are subsumed in the class *Management*. Depending on the complexity of the chosen strategies, a policy implementation may need more classes, which can then be added to the package.

As motivated in Section 4.4.3, the functional component `interceptor` does not contain additional TCB functions, but defines function signatures, which have to be implemented by a TCB's policies. For this reason, the corresponding *Interceptor* class is implemented as interface that is realized by the *RBACHISPolicy* class. The *Interceptor* is contained in an additional package, called *interceptors*, which provides the basis for building a hierarchy of interceptor classes within this package, in case a TCB contains many policies and possibly one or more meta policies. For executing a policy request, the implementation of the functional component `generic object manager` calls the interface methods of the *Interceptor* via a *uses*-dependency.

When combining this implementation of the policy-dependent RTE with an platform-specific implementation of the policy-independent RTE, additional *uses*-relations must emerge. In order that the *RBACHISPolicy* is correctly enforced in a TCB, it is reliant on the interface functions provided by the components of the policy-independent RTE. More precisely, a policy needs to call the methods of the implementation of the functional components `entity identification server`, `TPS manager`, `memory manager`, `thread manager`, and `transaction manager`.

6.4.2 Implementation Effort

Using the prototype implementation of Section 6.4.1, this section discusses the implementation effort that is involved whenever a policy requires to modify a TCB's functional perimeter. In doing so, we distinguish between three cases, each of which requiring a different implementation effort:

- Adding a Novel Policy,
- Removing a Policy, and
- Modifying a Policy.

Adding a Novel Policy The effort for modifying the implementation of a policy-dependent RTE to the requirements of a novel policy depends on the core-based model, by which the policy is expressed. Here, we must consider two possible cases as illustrated in Figure 6.2. (i) The model is a subsumed model and thus shares the definition of the state set Q and the extension vector E with at least one of the present policies. In this case, we do not need to add a new package, since the policy can be implemented by the functions of the existing packages. It only remains to check, whether the subsuming model already contains all primitive operations and clauses that are required by the novel policy, i.e., whether $\mathit{PRIM}_N \subseteq \mathit{PRIM}_S$ and $\mathit{CL}_N \subseteq \mathit{CL}_S$, where PRIM_N and CL_N are the sets of primitive operations and clauses of the novel policy's model and PRIM_S and CL_S are the corresponding sets of the subsuming model. If they are contained, the package that implements the subsuming model does not need to be modified at all. Otherwise, at least one primitive operation or clause is missing in the implementation such that corresponding functions must be added to the class *State_T* and/or *ExtVect_T*. (ii) The model is not subsumed by any model of the present policies and defines model-specific Q and E. Here, the implementation effort increases, since a new package, containing the classes to implement the model state, the extension vector, and the primitive types, must be added.

On top of that, we have to check in either of the two cases, whether the classes of the package *shared-model-functions* need to be modified. This is done by subtracting the set of derived TCB functions of all present policies $\mathcal{O}$ from the set of derived TCB functions of the novel policy $\mathcal{O}_N$. If the resulting set if empty, the methods that are contained in the classes of the *shared-model-functions* package are sufficient. Otherwise, we have to add new methods to the existing classes or we even have to implement new classes.

Removing a Policy In case a policy must be removed from a TCB, the implementation effort for adapting the policy-dependent RTE also depends on the model, by which the policy has been formalized. Here, the proceeding is similar to the one illustrated in Figure 6.2. The only differences are that implementation elements are removed rather than added and that the 2. and 3. distinction of cases, concerning the models' primitive actions (respectively clauses) and the set of derived TCB functions, are diametrically opposed.

More precisely, we first have to check whether the policy has been modeled by a subsumed model and hence shares a model package with other policies. If not, we can simply remove the package that belongs to the policy, including its classes and methods. Otherwise, we

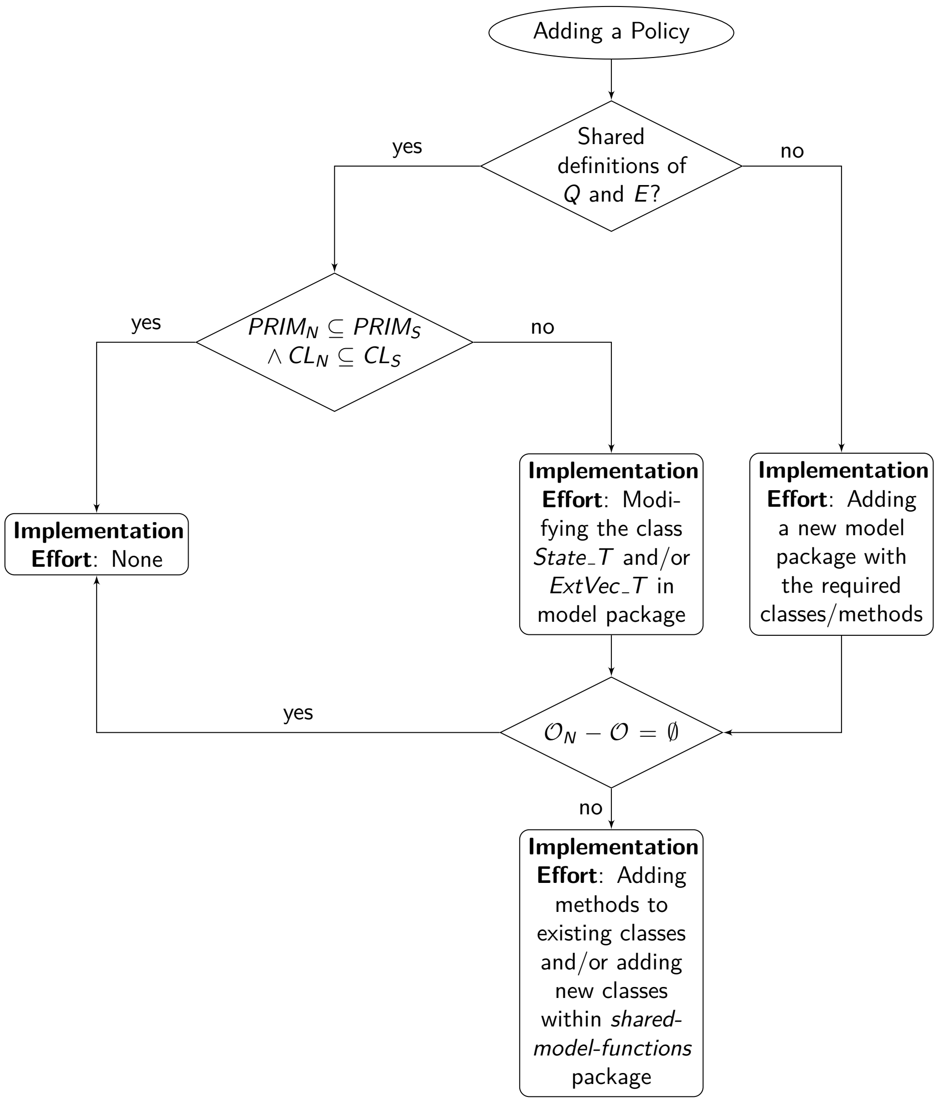
Adding a Policy
Shared definitions of Q and E?
yes
no
$PRIM_N \subseteq PRIM_S \wedge CL_N \subseteq CL_S$
yes
no
Implementation Effort: None
Implementation Effort: Modifying the class *State_T* and/or *ExtVec_T* in model package
Implementation Effort: Adding a new model package with the required classes/methods
$\mathcal{O}_N - \mathcal{O} = \emptyset$
yes
no
Implementation Effort: Adding methods to existing classes and/or adding new classes within *shared-model-functions* package

Figure 6.2: Implementation Effort for Adding a Novel Policy

have to check if the shared model package contains implementations of primitive operations or clauses that are not required by the subsumed models of the remaining policies. That means, we must verify if $\nexists prim \in PRIM_R : prim \notin PRIM_S$ and $\nexists cl \in CL_R : cl \notin CL_S$, where $PRIM_R$ and CL_R are the sets of primitive operations, respectively clauses, of the removed model and $PRIM_S$ and CL_S are the unified sets of the remaining subsumed models. If either one of the conditions is false, we have to remove the implementing method of *prim* or *cl* from the class *State_T* or *ExtVec_T*, since it now is superfluous. Here, we must be thorough not to delete other methods, since this would interfere with the RTE of the remaining policies.

Afterwards, we must make sure that superfluous functions do not remain in the *shared-model-functions* package. This is done by checking whether the implementation contains methods (due to the derived function set $\mathcal{O}_R$ of the removed policy) that are not contained in the unification of all derived TCB function sets $\mathcal{O}$ of the remaining policies, i.e., $\forall o \in \mathcal{O}_R : o \in \mathcal{O}$. If so, there are no superfluous TCB functions that have to be removed from the classes of the *shared-model-functions* package. Otherwise, we have to map any ADT operator $o \notin \mathcal{O}$

to its implementing method such that the method can be removed. In case all methods of a class are removed, the class can also me removed.

Modifying a Policy Basically, modifying a policy is identical to removing the functions of the old policy and adding the functions of the new one. However, there is one special case of policy modification, i.e., the modification of a policy's authorization scheme, which suggests another course of action in a real-world implementation. In contrast to the other ones, this policy modification is triggered by the *Interceptor*, who has changed its interface due to the supported OS and its applications. As a result the policy's authorization scheme must be modified to still implement the interface functions; however, all data of the policy, e.g., of the state and the extension vector, has to be maintained. Thus, the migration effort in a real-world implementation is reduced, if the existing policy-dependent RTE is modified instead of replaced.

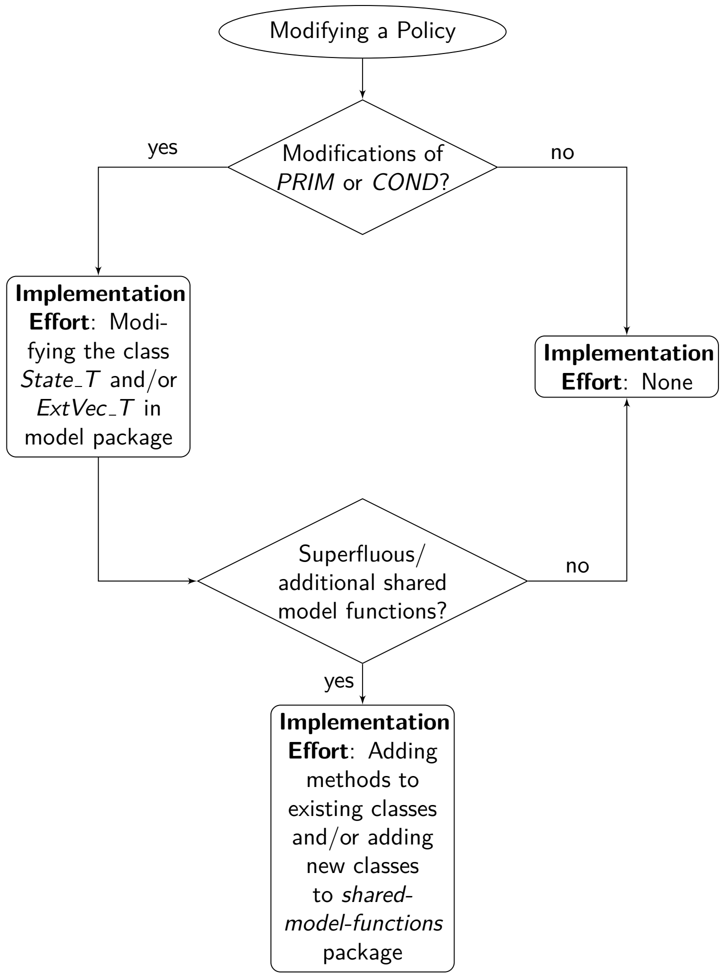
Modifying a Policy
Modifications of *PRIM* or *COND*?
yes
no
Implementation Effort: Modifying the class *State_T* and/or *ExtVec_T* in model package
Implementation Effort: None
Superfluous/ additional shared model functions?
no
yes
Implementation Effort: Adding methods to existing classes and/or adding new classes to *shared-model-functions* package

Figure 6.3: Implementation Effort for Modifying a Policy

The actual implementation effort depends on the changes made by the *Interceptor* though. In the simplest case, there is no need for modifications, since the modified commands of the authorization scheme still use the primitive operations and clauses that are already implemented. Otherwise, the methods of *State_T* and *ExtVec_T* must be modified, which may result in modifying the classes of the *shared-model-functions* package. The latter is necessary if either the modified primitive operations or clauses use additional set-theoretic operations that require additional TCB functions, or shared model functions become superfluous and thus have to be removed.

Summary This section has discussed the implementation effort that is involved whenever a policy requires to modify the implementation of a policy-dependent RTE. In all cases – adding, removing, and modifying a policy – the implementation effort actually is as low as discussed in the context of the functional TCB design in Section 4.2. The reason is that the other policies that are present in a TCB are not affected by any modifications if the necessary thoroughness is applied.

However, the implementation effort could be lower, if functional nonredundancy were not to be enforced in a TCB's implementation, too. The latter causes the distribution of the RTE functions so that a policy is implemented not only in model-specific packages, e.g., the *rbac-his-policy* package, but also in a *shared-model-functions* package. Thus, when policies are substituted, each location of the RTE functions must be analyzed with respect to missing or redundant TCB functions, thereby increasing the implementation effort.

Under consideration of this trade-off between functional nonredundancy and implementation effort, the latter is as low as possible, since other policies are not affected if the necessary thoroughness is applied. This sets the course for substituting policies in real-world application scenarios, where policies have to be enforced while obsolete policies are removed or novel policies are added.

6.5 Formal TCB Specifications

This section evaluates whether the specification method introduced in Section 5.4 produces formal TCB specifications that can serve as basis for generating an implementation of a TCB's policy-dependent RTE. That means, we have to analyze whether the resulting specifications meet the requirements discussed in Section 5.2. To this end, we have exemplarily applied the specification method to the RBAC HIS policy; the resulting TCB specification can be found in Appendix D. This section summarizes the results with respect of said requirements and the suitability of the employed specification language Event-B. An elaborate discussion of the results can be found in [178].

The RBAC HIS specification is *complete* and *externally consistent* with respect to the RBAC model instance because of the methodic approach that covers all model and model instance components. More precisely, the method has produced an isomorphism between the states and the state transitions of the model instance and the Event-B specification. However, we have encountered difficulties when proving the external consistency with an animator: the animation of the specification is only possible with increased effort since the employed ProB animator [239] cannot animate potentially infinite sets such as the state set in the model specification [178]. Besides enhancing the animator to circumvent this problem, an alternative approach is to further refine the specification towards individual state variables.

To assure *inner consistency*, we have employed mostly tool-supported validation techniques of the Rodin platform [7]. That way, the quality of the specification method significantly depends on the maturity of the employed tools as well as on the applied specification language. Here, the result is that the RBAC HIS specification contains more than 100 proof obligations, ca. 54% of which could be proven automatically, while the remaining ones had to been proven manually within the Rodin platform. This leads to a high degree of inner consistency; however, the specification effort can be reduced by an order of magnitude if the employed development platform is enhanced to prove more proof obligation types.

Reusability of the specifications is attained by separating the specification of the security model from a model instance's specification. That way, a model specification can be reused by any instance of the model. The degree of reusability could be higher though, if model specifications were able to reuse components of other model specifications. This requires a specification language that allows for building a hierarchy of model specifications based on extensive inheritance relations as briefly discussed in Section 8.

Since there are no metrics to evaluate the *size* and *complexity* of the RBAC HIS specification, we support our claim that this requirement is also met, by the following reasoning. The complexity of the RBAC HIS specification is located, as required, in the model specification, where the state set along with primitive operations is specified. A good part of the complexity arises from the necessity to specify the state set as a set instead of defining a general state machine for the model specification. This could be avoided by an improved specification language though. The main part of the complexity arises from specifying strict preconditions for the primitive operations, resulting in complex guards. The reason is that one state component (called `roles`) is defined as a power set, which increases the complexity of handling this component. This could also be avoided by remodeling the model component and its specification. With respect to the size of the specification, the initialization of the extension vector elements and the specification of the primitive operations are conspicuous. The reason for the latter are the validation theorems ($\mathtt{prim}_{\mathtt{i}}\mathtt{\_v}$), which are redundant regarding the primitives' definition theorems. To solve this redundancy without jeopardizing the consistency of definition and validation theorems, an improved specification language or additional tool support is required.

By this mean, we can reason for the necessity of an increased specification size and higher complexity based on the underlying model instance or the limits of Event-B. Meeting the requirement of small specifications with low complexity hence does not only depend on a thorough specification but also on the employed specification language and its development tools; and even though Event-B already provides sophisticated language features and tool support, both leaves room for improvements (Section 8).

In summary, the resulting TCB specification can serve as basis for generating the source code of the policy-dependent RTE due to the following reasons: (i) under consideration of the necessary thoroughness, the method generally produces formal specifications that meet all the requirements of Section 5.2. Thereby, TCB specifications are amenable to tool-supported validation techniques, which provides the foundation to stepwise generate TCB implementations that are amenable to verification. We have supported our claim by means of the TCB specification for the RBAC HIS policy. (ii) In [178] we have exemplarily shown how the formal specification of a model instance can be refined by using the derived ADTs of Section 4.5. The result is that the specification's level of abstraction becomes more implementation-oriented, while the correctness of each refinement step can be proved. (iii) There are a variety of industrial projects, which have demonstrated that generating source code based on Event-B specifications is feasible [71]. (iv) The Event-B community provides a set of plug-ins that support automated source code generation based on Event-B specifications [198]. The development of a compiler-like tool to generate the source code remains future work.

6.6 Summary

This section has evaluated causal TCB engineering with respect to the dissertation's main goal – the precise identification of a TCB's functional perimeter. Thereby, it has focused on ADT engineering, which produces the functions of a TCB's policy-dependent RTE.

A precisely defined functional perimeter of a policy-dependent RTE must be as small as possible with respect to a TCB's security policies, which in turn requires that it is nonredundant. This is met by ADT engineering by two engineering principles: functional subsumption and set union of TCB function sets. As a consequence, causal TCB engineering indeed is able to exactly define a causal TCB's functional perimeter. Furthermore, causal TCB engineering can be supported by an automated tool that composes the functional perimeter of a causal TCB, due to its sound theoretical foundation.

On top of that, this section has evaluated the model-based approach of causal TCB engineering with respect to its feasibility for real-world application scenarios. In this context, we have identified three additional requirements for TCB engineering, which are the corner pillars for unlocking causal TCBs for real-world application scenarios.

The first corner pillar requires that the common model core along with core-based model engineering can adequately express a wide variety of security models. While we have given proof that this requirement is met, we have also argued that the model engineering process is simplified, since the modeling effort is limited to specializing the components of the model core. Consequently, both – considerable expressive power as well as limited modeling effort – significantly contributes to unlocking causal TCBs to real-world applications.

The second corner pillar is a low implementation effort whenever the substitution of policies require to modify the implementation of a policy-dependent RTE. We have argued that by considering the required trade-off between functional nonredundancy and implementation effort, the latter is as low as possible, since other TCB policies are not affected if the necessary thoroughness is applied. This sets the course for substituting policies in real-world application scenarios, where policies must be enforced while obsolete policies are removed or novel policies are added.

The last corner pillar for unlocking causal TCBs for real-world application scenarios is method and tool support that bridges the gap between TCB functions and TCB implementations. This section has shown that the TCB specification method presented in Section 5.4 produces TCB specifications, which are amenable to tool-supported validation and can hence serve as basis for the automated generation of the source code of the policy-dependent RTE. By this means, the specification method promotes the efficiency of implementing a TCB's policy-dependent RTE while contributing the code's correctness and consistency with respect to the underlying model instance.

7 Conclusion

Competing is exciting and winning is exhilarating, but the true prize will always be the self-knowledge and understanding that you have gained along the way.

Sebastian Coe

Trusted Computing Bases (TCBs) of today's policy-controlled operating systems are characterized by a large functional perimeter. On the one hand, policy-controlled operating systems are based on commodity operating systems, thereby maintaining their monolithic architectures and scattering their policy enforcement mechanisms all over the kernel. On the other hand, the ambition of policy-controlled operating systems to provide support for a wide variety of security policies leads to universal policy decision and enforcement environments. Both results in large, complex, and expensive operating system TCBs whose functional perimeter can hardly be precisely identified. As a consequence, a TCB's essential properties – correctness, robustness, and tamper-proofness – are hard to ensure in a TCB's implementation.

This dissertation has followed a different approach based on the idea of systematically engineering TCBs by tailoring their policy decision and enforcement environment to support only those security policies that are actually present in a TCB. A TCB's functional perimeter is identified by exploiting causal dependencies between security policies and TCB functions, which results in causal TCBs that contain only those functions that are necessary to establish, enforce, and protect their policies. Causal TCB engineering sets the course for implementations whose size and complexity provide the basis for analyzing and verifying a TCB's correctness and tamper-proofness.

Security policies are usually described by informal sets of rules. Since these are inadequate for policy analyses and implementation, numerous formal security models have been developed that allow for formal analyses of security properties and increasingly serve as specifications for policy implementations. However, the wide variety of models makes it difficult to exploit common abstractions for causal TCB engineering. Thus, we have developed a formal uniform model foundation – which we refer to as security model core – that generalizes common model abstractions of access control and information flow models. Core-based model engineering then allows for creating a wide variety of dynamic security models with shared model abstractions, including identity-based, role-based, and attribute-based access control models (each in their discretionary and mandatory variant), as well as information flow and role-based trust management models; and model engineering is reduced to specializing the components of the model core. As a result, the model core provides the basis not only for causal TCB engineering but also for model-independent methods and tools for model engineering, analysis, and implementation.

In the context of causal TCB engineering, core-based models are used to derive a TCB's functions. The result is that all causal TCBs share common functions – contained in the policy-independent runtime environment (RTE) – to protect and enforce policies in core notation, and policy-dependent functions – composed in the policy-dependent RTE – that are derived due to causal dependencies between policies and TCB functions. We have identified the functional perimeter of the policy-independent RTE by motivating each function and the interface it provides. The identified functions are inspired by microkernel-based approaches and are redundancy-free. We also have developed a model-independent method for deriving the functions of the policy-dependent RTE based on an algebraic approach and a rule system that formalizes said causal dependencies. Functional redundancies are avoided by two approaches: subsuming TCB functions and set union of derived TCB function sets. The resulting TCB function sets are as small as possible with respect to a TCB's policies so that the functional perimeter of causal TCBs is precisely defined. On top of that, causal TCB engineering is based on a sound formal foundation, which allows for an automated TCB composition tool that composes a causal TCB's functional perimeter.

The challenge of implementing a causal TCB is to ensure implementation completeness and consistency with respect to the TCB functions. While the implementation of the policy-independent RTE has to be done only once, the policy-dependent RTE has to be implemented individually for each application scenario. For this reason, we have developed a specification method, independent of any specification language, that formally specifies a TCB's functions to serve as basis for automated source code generation of a TCB's policy-dependent RTE. The resulting TCB specifications are amenable to tool-supported validation techniques such that their completeness and consistency with respect to the TCB functions and the underlying security model can be ensured. This provides the foundation for generating TCB implementations that are amenable to verification. We have demonstrated this approach by configuring the specification method to Event-B, and we have applied the outcome to a role-based access control policy that is based on a real-world application scenario. The result is a TCB specification, whose consistency and correctness are validated by tool-supported validation techniques and methodic reasoning, and which can therefore serve as basis for source code generation. Thus, specification engineering improves not only the efficiency but also the effectiveness of generating a TCB's implementation.

In summary, this dissertation has developed a model-based TCB engineering method that contributes to a comprehensive security policy engineering process model without semantic gaps, whose efficiency and effectiveness are increased by model-independent methods and tools for model engineering, analysis, TCB engineering, and policy implementation. To the best of our knowledge, this dissertation is the first approach that identifies a TCB's components already at the functional level. The precise identification of a TCB's functional perimeter allows for implementing a TCB in a safe environment that indeed can be isolated from untrusted system components. On top of that, it defines the scope of system verification. Thereby, causal TCB engineering sets the course for implementations whose size and complexity provide the basis for analyzing and verifying a TCB's correctness and tamper-proofness. The application scenarios for causal TCB engineering range from embedded systems and policy-controlled operating systems to database management systems in large information systems.

8 Future Work

To be conscious that you are ignorant is a great step to knowledge.

Benjamin Disraeli,
Sybil, 1845

This dissertation has developed an engineering method for causal TCBs. The unique feature of causal TCBs is a precisely identified functional perimeter, which supports only those security policies that are actually present in a TCB. The foundation of causal TCB engineering is a unified formalization of the policies using security models in core notation.

Once a policy is formalized as a core-based model instance, it is unlocked not only to causal TCB engineering but also to a family of model-independent analyzing methods and tools. In [16] we have introduced one member of this family: a general heuristic-based analysis method for access control models, which we have prototypically configured to the HRU model [107]. We also have presented a security policy engineering workbench, whose main component is a heuristic-based symbolic model execution engine. Future work aims at configuring this analysis method to models that can be applied to real-world policies, thereby enhancing the workbench. As a first step, we have started the WorSE project, which configures this method to dynamic RBAC models in core notation that are based on Sandhu's RBAC model family [205]. Depending on the results, this approach may also be relevant for more current and sophisticated models such ABAC [251], parametrized RBAC [99, 228], or GEO-RBAC [31].

In Section 4.1.2 we have argued that multi-policy support is an important feature of causal TCBs, which requires policy isolation, communication, and coordination. While causal TCBs already provide mechanisms for policy isolation and basic mechanisms for policy communication, policy coordination has not been considered. Future work may tackle this problem beginning at the level of model engineering to the point of TCB engineering. To model policy coordination either by policy composition or meta policies, the core notation could be enhanced, which has the significant advantage that necessary TCB functions can afterwards be derived by ADT engineering. For example, meta policies could be modeled as a hierarchy of core-based security models. On the other hand, policy composition needs to express composition rules for policy decisions.

The next step towards multi-policy support is at the level of causal TCB engineering. Here, the policy-independent RTE must be enhanced by a new functional component for policy coordination (e.g., called `policy scheduler`) and mechanism for *inter policy process communication.* The `policy scheduler` then provides the abstraction `meta policies`, which manages the policies that are involved in common policy decisions based on application-specific strategies. On the other hand, inter policy process communication enables policy composition by explicitly allowing `security policy processes` to communicate with each other.

Another open problem at the level of TCB engineering is tool support for composing a causal TCB's functional perimeter. We have already provided the theoretical foundation by following an algebraic approach and defining a formal rule system (Section 4). It remains to develop an automated composition tool that implements the formal rule system. Input of this tool is a core-based security model in algebraic notation; output is a set of related ADTs. To tackle this problem, compiler-construction tools like *Flex* and *Bison* [151], *Memphis* [2], or *GENTLE* [214, 215] can be applied that support a developer by generating a lexical analyzer and a syntactic parser, or by providing means to manipulate symbolic data [1].

As discussed during the evaluation of the TCB specification method for Event-B (Section 6.5), the specification method can be further improved by a domain-specific specification language. For example, one drawback of Event-B is that it does not support specification hierarchies. In this case, specification engineering would be similar to core-based model engineering: first, the common model core is specified from which specifications of security models can be derived. Second, TCB specifications are created by deriving the specifications of model instances from a model specification. Hence, the degree of reusability of TCB specifications could be further increased. Though, it is not necessary to develop a novel specification language. Due to the extensibility of Event-B and its tools, new language features can be designed by integrating them in Event-B and its development platform. As a result, Event-B tools such as the theorem proofer and the proof obligation generator can still be used.

It then remains to develop a compiler-like tool that based on a validated formal TCB specification generates a correct implementation, which is amenable to verification. In doing so, future work can benefit from existing case studies, tool support, and refinement and transformation techniques.

Another open problem for future work is the implementation of causal TCBs. More precisely, while Section 6.4 has focused on implementing a specific policy-dependent RTE, it remains future work to implement the policy-independent RTE on a specific platform. Since the functional design of the policy-independent RTE is mainly inspired by microkernels, we suggest to implement it using a microkernel such as *L4* [162, 163], which has been implemented by several systems, e.g., *Fiasco.OC* [236] or the formally verified *seL4* [137, 138]. Section 2.2.1 has introduced Nizza [123] as a security architecture built on top of Fiasco that aims at isolating trusted from untrusted source code while providing the functionality of a standard OS (Linux). Implementing causal TCBs within the Nizza architecture thus has the advantage of reusing existing Nizza technologies, e.g., Fiasco, L^{4}Linux, or authenticated booting, along with a sophisticated development platform called L4RE [237]. Nizza's reference implementation aims at a minimal amount of trusted code so that we have a sound basis for implementing causal TCBs. Based on these observations, we have started a new project that has already made the first step towards implementing causal TCBs within the Nizza architecture by analyzing Nizza's isolation mechanisms for `policy process` isolation. The goal is to implement some components of the policy-independent RTE and the complete policy-dependent RTE as trusted service within the secure-platform layer. The remaining components of the policy-independent runtime environment are already implemented in Fiasco.

A Security Models

A.1 Notation

Relations In this dissertation, we consider Relations as sets of tuples. Given two sets A and B, $R \subseteq A \times B$ is a relation where $A' \subseteq A$ and $B' \subseteq B$. A is called domain and B is called range of R.

The following definitions are based on the notation of formal specifications [72].

Domain Restriction $A' \lhd R := \{(a, b) \in R \mid a \in A'\}$
Domain restriction $A' \lhd R$ of a relation R means that the domain A of R is restricted to the subset A'.

Range Restriction $R \rhd B := \{(a, b) \in R \mid b \in B'\}$
Range restriction $R \rhd B$ of a relation R means that the range B of R is restricted to the subset B'.

Domain Subtraction $A' \trianglelefteq R := (A \backslash A') \lhd R = \{(a, b) \in R \mid a \notin A'\}$
Domain subtraction $A' \trianglelefteq R$ of a relation R means that the subset A' is subtracted from the domain A of R.

Range Subtraction $R \trianglerighteq B' := R \rhd (B \backslash B') = \{(a, b) \in R \mid b \notin B'\}$
Range subtraction $R \trianglerighteq B'$ of a relation R means that the subset B' is subtracted from the range B of R.

Functions and Functional Overriding Functions are considered as left-total and right-unique relations such that for two functions f and g holds, $f \cup g$ is also a function when $\forall x \in dom(f) \cap dom(g) : f(x) = g(x)$. Given two functions $f : X \to Z$ and $g : Y \to Z$. The overriding of f by g is defined as $f \oplus g : (X \cup Y) \to Z$ and $f \oplus g := ((X \backslash Y) \lhd f) \cup g = (Y \trianglelefteq f) \cup g$.

A.2 MLS Model

Based on the security model of Hutter et al. [118–120], the security model for the web service composition policy is defined as $(Q, \Sigma, \delta, q_0, (LO, FI, \leqslant, O), \lambda)$ where

- $Q = 2^{WS} \times CL \times DEL$ is the set of states and any state q is defined as $q = (WS_q, cl_q, del_q)$ with $WS_q \subseteq WS$, $cl_q \in CL$, and $del_q \in DEL$, where
  - WS is the infinite set of web services,
  - $CL = \{cl \mid cl : WS \cup O \to LO \times FI\}$ is the classification function that assigns a security class to a web service/object, and

  - $DEL = \{del \mid del : WS \to LO \times FI\}$ is the delegation function that assigns a security class to a web service.

- $\Sigma = C \times X$ is the set of inputs where C is the policy-specific set of commands and $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$ is the set of l-dimensional input vectors and $T = \{2^{WS}, CL, DEL, WS, cl, del, LO, FI, \leqslant, O, 2^{LO}, 2^{FI}, 2^{\leqslant}, 2^{O}\}$.
- δ is the state transition function $\delta : Q \times \Sigma \to Q$ with conditions and primitive actions as defined below, and
- $q_0 \in Q$ where $q_0 = (WS_0, cl_0, del_0)$ is the initial state.
- $(LO \times FI, \leqslant, O)$ is the extension vector of static model components where
  - LO and FI are the security classes with $(lo_1, fi_2) \leqslant (lo'_1, fi'_2)$ iff $lo_1 \leqslant lo'_2$ and $fi_1 \leqslant fi'_2$ holds, and
  - O is the finite set of objects,
- $\lambda : Q \times (A \cup C) \times X \to \{true, false\}$ is the output function and A is the set of application specific operations.

The set of conditions $COND$ consists of three elements, where $cond_{MLS_1}$ represents one of two BLP conditions checking whether information flow is allowed from objects to web services. The second condition of BLP ($cl(ws) \leqslant cl(o)$) is implemented by $cond_{MLS_2}$. The condition $cond_{DEL}$ checks whether copying the delegation classification of one web service to a new one is allowed. It holds true as long as the delegating web service is not classified the lowest possible security class D_1 for LO and D_2 for FI.

$$cond_{MLS_1} : Q \times X \to BOOL \text{ where}$$

$$cond_{MLS_1}(q, x_{ws}, x_o) \mapsto \begin{cases} true, & clause_{MLS_1}(cl_q(x_o), cl_q(x_{ws})) \\ false, & \text{otherwise} \end{cases}$$

$$cond_{MLS_2} : QX \to BOOL \text{ where}$$

$$cond_{MLS_2}(q, x_{ws}, x_o) \mapsto \begin{cases} true, & clause_{MLS_2}(cl_q(x_o), cl_q(x_{ws})) \\ false, & \text{otherwise} \end{cases}$$

$$cond_{DEL} : Q \times X \to BOOL \text{ where}$$

$$cond_{DEL}(q, x_{ws}) \mapsto \begin{cases} true, & clause_{DEL}(q, cl_q(x_{ws})) \\ false, & \text{otherwise} \end{cases}$$

Thus, a total of three clauses is needed to model these conditions:

$$clause_{MLS_1} : Q \times X \to BOOL$$

$$clause_{MLS_1}(q, x_l, x_f) \mapsto \begin{cases} true, & x_l \leqslant x_f \\ false, & \text{otherwise} \end{cases}$$

$$clause_{MLS_2} : Q \times X \to BOOL$$

$$clause_{MLS_2}(q, x_l, x_f) \mapsto \begin{cases} true, & x_f \leqslant x_l \\ false, & \text{otherwise} \end{cases}$$

$$clause_{DEL} : Q \times X \rightarrow BOOL$$

$$clause_{DEL}(q, x_l, x_f) \mapsto \begin{cases} true, & x_l \neq `D_1` \wedge x_f \neq `D_2` \\ false, & \text{otherwise} \end{cases}$$

The set of primitives *PRIM* consists of for elements. According to the policy where web services can only be added and delegations and classifications can only be granted once, primitives for deleting web services and for reclassify web services and objects are not required. Thus, the security model contains three primitives (*addWebService, classify, setDelegation*) for adding web services and classifying web service and objects. Additionally, a fourth primitive action *copyClassification* enables a web service to copy its delegation classification to a new web services that is added to the composition plan.

$$addWebService : Q \times X \rightarrow Q$$
$$addWebservice(q, x_{ws_1}, ..., x_{ws_n}) \mapsto (WS_q \cup \{x_{ws_1}, ..., x_{ws_n}\}, cl_q, del_q)$$

$$classify : Q \times X \rightarrow Q$$
$$classify(q, x_e, x_l, x_f) \mapsto (WS_q, cl_q \oplus \{(x_e, (x_l, x_f))\}, del_q)$$

$$setDelegation : Q \times X \rightarrow Q$$
$$setDelegation(q, x_e, x_l, x_f) \mapsto (WS_q, cl_q, del_q \oplus \{(x_e, (x_l, x_f))\})$$

$$copyClassification : Q \times X \rightarrow Q$$
$$copyClassification(q, ws_{old}, ws_{new}) \mapsto (WS_q, cl_q \oplus \{(ws_{new}, del_q(ws_{old}))\}, del_q)$$

A.3 RBAC Model

Based on the RBAC$_3$ security model of Sandhu's RBAC model family [86, 87, 205, 210, 212], the core-based RBAC security model for the health care policy of Section 3.4.4 is defined as a tuple $(Q, \Sigma, \delta, q_0, (O, OP, R, m, RH, RE), \lambda)$ where:

- $Q = 2^U \times 2^S \times 2^{UA} \times USER \times ROLES$ is the set of states and any state q is defined as $q = (U_q, S_q, UA_q, user_q, roles_q)$ with $U_q \subseteq U, S_q \subseteq S, UA_q \subseteq UA, user_q \in USER$, and $roles_q \in ROLES$, where
  - U is the set of users,
  - S is the set of sessions,
  - $UA \subseteq U \times R$ is the user-to-role assignment relation,
  - $USER = \{user \mid user : S \rightarrow U\}$ is the user-to-session association function that maps a session onto the session's associated user, and
  - $ROLES = \{roles \mid roles : S \rightarrow 2^R\}$ is the session-roles-activation function mapping a session onto a subset of its associated user's roles, called activation of roles.
- $\Sigma = C \times X$ is the set of inputs where C is the policy specific set of commands and $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$ is the set of l-dimensional input vectors and $T = \{2^U, 2^S, 2^{UA}, USER, ROLES, U, S, UA, user, roles, O, OP, R, m, RH, RE, 2^O, 2^{OP}, 2^R, 2^m, 2^{RE}\}$.
- δ is the state transition function $\delta : Q \times \Sigma \rightarrow Q$ with conditions and primitive actions as defined below, and

- $q_0 \in Q$ where $q_0 = (U_0, S_0, UA_0, user_0, roles_0)$ is the initial state.
- (O, OP, R, m, RH, RE) is the extension vector of static model components where
  - O is the finite set of objects,
  - OP is the finite set of operations,
  - R is the finite set of roles,
  - $m : R \times O \to 2^{OP}$ is the role-to-permission assignment matrix,
  - $RH \subseteq R \times R$ is a partial order on R called the role hierarchy relation, written as $\geq$, where $\forall r, r' \in R : r \geq r' \Leftrightarrow$ all permissions of r' are also permissions of r and all users of r are also users of r' [86],
  - $RE \subseteq R \times R$ is a nonreflexive, symmetric, and nontransitive relation on R, written as $\approx_{ex}$, and called as role exclusion relation where $\forall r, r' \in R : r \approx_{ex} r' \Leftrightarrow r$ and r' are mutually exclusive.
- $\lambda : Q \times (A \cup C) \times X \to \{true, false\}$ is the output function where A is the set of application specific operations.

The set of conditions $COND$ consists of five elements, which are defined as follows:

$$cond_{core} : Q \times X \to BOOL$$

$$cond_{core}(q, x_s, x_o, x_{op}) \mapsto \begin{cases} true, & \exists r, r' \in R : clause_{RH}(r', r) \\ & \wedge\ clause_{roles}(q, x_s, r') \ \wedge\ clause_m(x_{op}, r, x_o) \\ false, & \text{otherwise} \end{cases}$$

$$cond_{UM} : Q \times X \to BOOL$$

$$cond_{UM}(q, x_u, x_o, x_{op}) \mapsto \begin{cases} true, & \exists r, r' \in R : clause_{RH}(r', r) \\ & \wedge\ clause_{UA}(q, r', x_u) \ \wedge\ clause_m(x_{op}, r, x_o) \\ false, & \text{otherwise} \end{cases}$$

$$cond_{SR} : Q \times X \to BOOL$$

$$cond_{SR}(q, x_s, x_r) \mapsto \begin{cases} true, & \exists r' \in R : clause_{RH}(r', x_r) \ \wedge\ clause_{roles}(q, x_s, r') \\ false, & \text{otherwise} \end{cases}$$

$$cond_{UR} : Q \times X \to BOOL$$

$$cond_{UR}(q, x_u, x_r) \mapsto \begin{cases} true, & \exists r' \in R : clause_{RH}(r', x_r) \ \wedge\ clause_{UA}(q, r', x_u) \\ false, & \text{otherwise} \end{cases}$$

$$sod : Q \times X \to BOOL$$

$$sod(q, x_u, x_r) \mapsto \begin{cases} true, & \forall r' \in R : clause_{RE}(x_r, r') \ \wedge\ clause_{UA}(q, r', x_u) \\ false, & \text{otherwise} \end{cases}$$

It can be seen that all conditions are based on a total of five clauses. These are defined in the following:

$$clause_{roles} : Q \times X \to BOOL$$

$$clause_{roles}(q, x_s, x_r) \mapsto \begin{cases} true, & x_r \in roles_q(x_s) \\ false, & \text{otherwise} \end{cases}$$

$$clause_{UA} : Q \times X \to BOOL$$

$$clause_{UA}(q, x_r, x_u) \mapsto \begin{cases} true, & (x_u, x_r) \in UA_q \\ false, & \text{otherwise} \end{cases}$$

$$clause_m : X \to BOOL$$

$$clause_m(x_{op}, x_r, x_o) \mapsto \begin{cases} true, & x_{op} \in m(x_r, x_o) \\ false, & \text{otherwise} \end{cases}$$

$$clause_{RH} : X \to BOOL$$

$$clause_{RH}(x_{r_1}, x_{r_2}) \mapsto \begin{cases} true, & x_{r_1} \geq x_{r_2} \\ false, & \text{otherwise} \end{cases}$$

$$clause_{RE} : X \to BOOL$$

$$clause_{RE}(x_{r_1}, x_{r_2}) \mapsto \begin{cases} true, & \neg(x_{r_1} \approx_{ex} x_{r_2}) \\ false, & \text{otherwise} \end{cases}$$

The *access*-function of standard RBAC security models [86] exactly defines the constraints of role-based access control decisions. Thus, it is required by our role-based RBAC model and therefore rewritten as condition $cond_{core}$. One precondition of $cond_{core}$ is that a user is already associated with a session having activated a subset of the user's roles. However, this precondition cannot be met, as long as a user is not associated with a session, i.e., as long as the user is not logged in. Logging in, however, modifies a policy's state. Hence, the security model must provide a condition such that a policy is able to avoid its modification when necessary: $cond_{UM}$.

Real-world RBAC policies like the health information system policy given in Section 3.4.4 frequently require that users are assigned specific roles, without considering the roles' permission, e.g., for administration purposes. This approach is usually applied when policies are based on static role sets and role-to-permission assignments. In order to model these real-world requirements, our core-based RBAC model provides the conditions $cond_{UR}$ and $cond_{SR}$ which allow for checking whether a user or a session is assigned a specific role. For these conditions the role's permissions are irrelevant.

Finally, condition *sod* implements mutual exclusion of roles. This condition has to be part of every policy-specific command modifying UA by inserting new pairs (u, r) (implemented by primitive *assignUserToRoles*).

The set of primitive actions $PRIM$ contains ten elements that perform pairwise modifications of the state elements:

$$addUsers : Q \times X \to Q$$

$$addUsers(q, \{x_{u_1}, ..., x_{u_n}\}) \mapsto (U_q \cup \{x_{u_1}, ..., x_{u_n}\}, S_q, UA_q, user_q, roles_q)$$

$$\begin{aligned}
&\mathit{deleteUsers} : Q \times X \to Q\\
&\mathit{deleteUsers}(q, \{x_{u_1}, ..., x_{u_n}\}) \mapsto (U_q \backslash \{x_{u_1}, ..., x_{u_n}\}, S_q, \{x_{u_1}, ..., x_{u_n}\} \lhd UA_q,\\
&\qquad\qquad \mathit{user}_q \rhd \{x_{u_1}, ..., x_{u_n}\}, \mathit{roles}_q)\\
&\mathit{createSessions} : Q \times X \to Q\\
&\mathit{createSessions}(q, \{x_{s_1}, ..., x_{s_n}\}) \mapsto (U_q, S_q \cup \{x_{s_1}, ..., x_{s_n}\}, UA_q, \mathit{user}_q,\\
&\qquad\qquad \mathit{roles}_q \oplus \{(x_{s_1}, \varnothing), ..., (x_{s_n}, \varnothing)\})\\
&\mathit{destroySessions} : Q \times X \to Q\\
&\mathit{destroySessions}(q, \{x_{s_1}, ..., x_{s_n}\}) \mapsto (U_q, S_q \backslash \{x_{s_1}, ..., x_{s_n}\}, UA_q, \{x_{s_1}, ..., x_{s_n}\} \lhd \mathit{user}_q,\\
&\qquad\qquad \{x_{s_1}, ..., x_{s_n}\} \lhd \mathit{roles}_q)\\
&\mathit{mapUserSessions} : Q \times X \to Q\\
&\mathit{mapUserSessions}(q, x_u, \{x_{s_1}, ..., x_{s_n}\}) \mapsto (U_q, S_q, UA_q, \mathit{user}_q \oplus \{(s_1, x_u), ..., (s_n, x_u)\},\\
&\qquad\qquad \mathit{roles}_q)\\
&\mathit{unmapUserSessions} : Q \times X \to Q\\
&\mathit{unmapUserSessions}(q, x_u, \{x_{s_1}, \ldots, x_{s_n}\}) \mapsto (U_q, S_q, UA_q,\\
&\qquad\qquad \mathit{user}_q \backslash \{(x_{s_1}, x_u), ..., (x_{s_n}, x_u)\}, \mathit{roles}_q)\\
&\mathit{assignUserToRoles} : Q \times X \to Q\\
&\mathit{assignUserToRoles}(q, x_u, \{x_{r_1}, ..., x_{r_n}\}) \mapsto (U_q, S_q, UA_q \cup \{(x_u, x_{r_w}), ..., (x_u, x_{r_n})\},\\
&\qquad\qquad \mathit{user}_q, \mathit{roles}_q)\\
&\mathit{revokeUserFromRoles} : Q \times X \to Q\\
&\mathit{revokeUserFromRoles}(q, x_u, \{x_{r_1}, ..., x_{r_n}\}) \mapsto (U_q, S_q, UA_q \backslash \{(x_u, x_{r_w}), ..., (x_u, x_{r_n})\},\\
&\qquad\qquad \mathit{user}_q, \mathit{roles}_q)\\
&\mathit{activateRoles} : Q \times X \to Q\\
&\mathit{activateRoles}(q, x_s, \{x_{r_1}, ..., x_{r_n}\}) \mapsto (U_q, S_q, UA_q, \mathit{user}_q,\\
&\qquad\qquad \mathit{roles}_q \oplus \{(x_s, \mathit{roles}_q(x_s) \cup \{x_{r_1}, ..., x_{r_n}\})\}),\\
&\mathit{deactivateRoles} : Q \times X \to Q\\
&\mathit{deactivateRoles}(q, x_s, \{x_{r_1}, ..., x_{r_n}\}) \mapsto (U_q, S_q, UA_q, \mathit{user}_q,\\
&\qquad\qquad \mathit{roles}_q \oplus \{(x_s, \mathit{roles}_q(x_s) \backslash \{x_{r_1}, ..., x_{r_n}\})\})
\end{aligned}$$

A.4 ABAC Model

Rewriting the ABAC model for web services of Yuan and Tong [251] for their Online Entertainment Store example inspired by [10] results in a core-based ABAC model that is defined by a tuple $(Q, \Sigma, \delta, q_0, (SA, RA), \lambda)$ as follows:

- $Q = 2^S \times 2^R \times A_S \times A_R$ is the set of states and any state q is defined as $q = (S_q, R_q, a_{S_q}, a_{R_q})$ with $S_q \subseteq S, R_q \subseteq R, a_{S_q} \in A_S$, and $a_{R_q} \in A_R$, where
  - S is the set of subjects,
  - R is the set of resources,
  - $A_S = \{a_S \mid a_S : S \to AGE \times ROLE\}$ is the attribute assignment mapping for subjects that maps a subject onto its attributes, and

  - $A_R = \{a_R \mid a_R : R \to \mathit{RATING} \times \mathit{TYPE}\}$ is the attribute assignment mapping for resources that assigns each object to its attributes.

- $\Sigma = C \times X$ is the set of inputs where C is the policy specific set of commands and $X = \{(x_0, ..., x_l) \mid \forall x_i, 1 \leqslant i \leqslant l : x_i \in t_j, t_j \in T, 0 \leqslant j < |T|\}$ is the set of l-dimensional input vectors and $T = \{2^S, 2^R, A_S, A_R, S, R, a_S, a_R, \mathit{AGE}, \mathit{ROLE}, \mathit{RATING}, \mathit{TYPE}, 2^{\mathit{AGE}}, 2^{\mathit{ROLE}}, 2^{\mathit{RATING}}, 2^{\mathit{TYPE}}\}$.
- δ is the state transition function $\delta : Q \times \Sigma \to Q$ with conditions and primitive actions as defined below, and
- $q_0 \in Q$ where $q_0 = (S_0, R_0, a_{S_0}, a_{R_0})$ is the initial state.
- $(\mathit{SA}, \mathit{RA})$ is the extension vector of static model components where
  - SA is the finite set of subject attribute sets where $\mathit{SA} = \{\mathit{AGE}, \mathit{ROLE}\}$,
  - RA is the finite set of resource attribute sets where $\mathit{RA} = \{\mathit{RATING}, \mathit{TYPE}\}$.
- $\lambda : Q \times (A \cup C) \times X \to \{\mathit{true}, \mathit{false}\}$ is the output function where A is the set of application specific operations.

In contrast to the other core-based security models presented in this dissertation, the ABAC model is tailored for a specific policy rather than for a set of policies. This becomes obvious when defining the model's clauses. Here, it is necessary to predefine the subject and resource attributes since they are required in the clauses. This is specific for this model; for the MLS and the RBAC model it has been sufficient to define the extension vector as part of policy engineering. AGE represents the age of subjects, thus $\mathit{AGE} \subseteq \mathbb{N}$ where $\mathit{AGE} = \{1, \ldots, 150\}$ should be sufficient. ROLE contains the roles that a subject may act upon. While the example already defines the roles *Premium* and *Regular* for premium and regular customers, an additional role *Admin* is required for policy management. RATING represents the movie ratings $R, PG - 13$, and G. Moreover, movies are categorized either as *NewRelease* or *OldRelease* based on release dates, which are the values of the resource attribute TYPE.

The set of conditions COND consists of two elements, where $\mathit{cond}_{\mathit{access}}$ is required to check whether a subject may access a given movie and $\mathit{cond}_{\mathit{man}}$ is required to check whether management operations are allowed:

$$\mathit{cond}_{\mathit{access}} : Q \times X \to \mathit{BOOL}$$
$$\mathit{cond}_{\mathit{access}}(q, x_s, x_r) \mapsto \begin{cases} \mathit{true}, & \mathit{clause}_{AR}(q, x_s, x_r) \wedge \mathit{clause}_{RT}(q, x_s, x_r) \\ \mathit{false}, & \text{otherwise} \end{cases}$$

$$\mathit{cond}_{\mathit{man}} : Q \times X \to \mathit{BOOL}$$
$$\mathit{cond}_{\mathit{man}}(q, x_s, x_r) \mapsto \mathit{clause}_{\mathit{man}}(q, x_s)$$

To implement these conditions, three clauses are required. While clause_{AR} implements the policy rule $R3$ defined by the original ABAC model [251], clause_{RT} implements the policy rule $R4$. $\mathit{clause}_{\mathit{man}}$ is not contained in the original model, since it does not deal with policy administration. However, the core-based ABAC model for the Online Entertainment Store example requires a clause to distinguish ordinary users from privileged users for policy administration. Note that we use the functional notation $\mathit{age}_q(s)$, $\mathit{role}_q(s)$, $\mathit{rat}_q(r)$, and

$type_q(r)$ to select value assignments of individual attributes of the attribute tuples of s and r for a specific state q.

$$clause_{AR} : Q \times X \to BOOL$$

$$clause_{AR}(q, x_s, x_r) \mapsto \begin{cases} true, & (age_q(x_s) \geqslant 21 \wedge rat_q(x_r) \in \{R, PG, G\}) \vee \\ & (age_q(x_s) \geqslant 13 \wedge age_q(x_s) < 21 \wedge rat_q(x_r) \in \{PG, G\}) \\ & \vee\ (age_q(x_s) < 13 \wedge rat_q(x_r) \in \{G\}) \\ false, & \text{otherwise} \end{cases}$$

$$clause_{RT} : Q \times X \to BOOL$$

$$clause_{RT}(q, x_s, x_r) \mapsto \begin{cases} true, & (role_q(x_s) = \text{`}Premium\text{'}) \vee \\ & (role_q(x_s) = \text{`}Regular\text{'} \wedge type_q(x_r) = \text{`}OldRelease\text{'}) \\ false, & \text{otherwise} \end{cases}$$

$$clause_{man} : Q \times X \to BOOL$$

$$clause_{man}(q, x_s) \mapsto \begin{cases} true, & role_q(x_s) = \text{`}Admin\text{'} \\ false, & \text{otherwise} \end{cases}$$

The sets of primitive actions consists of eight elements, which pairwise modify the model's state components.

$$addSubject : Q \times X \to Q$$
$$addSubject(q, \{x_s\}) \mapsto (S_q \cup \{x_s\}, R_q, a_{S_q}, a_{R_q})$$

$$deleteSubject : Q \times X \to Q$$
$$deleteSubject(q, \{x_s\}) \mapsto (S_q \backslash \{x_s\}, R_q, \{s\} \lhd a_{S_q}, a_{R_q})$$

$$addResource : Q \times X \to Q$$
$$addResource(q, \{x_r\}) \mapsto (S_q, R_q \cup \{x_r\}, a_{S_q}, a_{R_q})$$

$$deleteResource : Q \times X \to Q$$
$$deleteResource(q, \{x_r\}) \mapsto (S_q, R_q \backslash \{x_r\}, a_{S_q}, \{r\} \lhd a_{R_q})$$

$$assignSubjAttributes : Q \times X \to Q$$
$$assignSubjAttributes(q, x_s, x_{age}, x_{role}) \mapsto (S_q, R_q, a_{S_q} \oplus \{(x_s, x_{age}, x_{role})\}, a_{R_q})$$

$$resignSubjAttributes : Q \times X \to Q$$
$$resignSubjAttributes(q, x_s, x_{age}, x_{role}) \mapsto (S_q, R_q, a_{S_q} \backslash \{(x_s, x_{age}, x_{role})\}, a_{R_q})$$

$$assignObjAttributes : Q \times X \to Q$$
$$assignSubjAttributes(q, x_r, x_{rating}, x_{type}) \mapsto (S_q, R_q, a_{S_q}, a_{R_q} \oplus \{(x_r, x_{rating}, x_{type})\})$$

$$resignObjAttributes : Q \times X \to Q$$
$$resignObjAttributes(q, x_r, x_{rating}, x_{type}) \mapsto (S_q, R_q, a_{S_q}, a_{R_q} \backslash \{(x_s, x_r, x_{rating}, x_{type})\})$$

B Health Information System Model Instance

The security policy used in this dissertation is roughly based on a small real-world Health Information System (HIS) for an aged-care facility and was first introduced by [77]. In order to develop a formal model for this policy, [99, 228, 229] have precisely specified the policy, enhanced it by additional policy rules, and extended the policy with parameters, or some aspects of the electronic health records policy [25]. We have illustrated a core-based RBAC model $(Q, \Sigma, \delta, q_0, (O, OP, R, m, RH, RE), \Omega, \lambda)$ in Appendix A.3 that provides suitable model abstractions for the RBAC HIS policy.

On this account, this appendix introduces an instance of the core-based RBAC model formalizing the RBAC HIS policy. Due to demonstration purposes, we present only a small excerpt of the policy rules, containing ten roles, 15 objects, seven operations, a role hierarchy, and separation-of-duty based on role exclusion. Additionally, we have added a collection of user management commands (e.g., *createUser(), destroyUser()*) including a role *UserAdmin*, which is dedicated to manage the users and their roles.

We first present the initial state and extension values of the model instance (Appendix B.1) before modeling the policy's authorization scheme (Appendix B.2)

B.1 Initial State and Extension Values

The initial state $q_0 = (U_0, S_0, UA_0, user_0, roles_0)$ contains a user u_1, representing a privileged instance, which is assigned the role *UserAdmin*. Otherwise, the initial state is defined as follows:

$$
\begin{aligned}
&U_0 = \{u_1\}\\
&S_0 = \varnothing\\
&UA_0 = \{\,(u_1, \mathit{UserAdmin})\,\}\\
&\mathit{user}_0 = \varnothing\\
&\mathit{roles}_0 = \varnothing
\end{aligned}
$$

The security model contains the extension vector (O, OP, R, m, RH, RE) that consists of six static model components. We initialize each of the static components in the following. The set of objects O contains the objects of the Health Information System; for this use case scenario, we have selected the most important objects.

$$
\begin{aligned}
O = \{\,&\mathit{OldMedicalRecords}, \mathit{RecentMedicalRecords}, \mathit{PrivateNotes}, \mathit{Prescriptions},\\
&\mathit{PatientMedicalInfo}, \mathit{PatientPersonalInfo}, \mathit{PatientFinancialInfo}, \mathit{CarePlan},\\
&\mathit{Appointment}, \mathit{ProgressNotes}, \mathit{LegalAgreement}, \mathit{Bills}, UA_o, U_o\}
\end{aligned}
$$

Medical records are records that a doctor adds an entry to after each examination [77]. *OldMedicalRecords* are records of former examinations; *RecentMedicalRecords* contain current examination results respectively. *PrivateNotes* contain personal notes of a doctor. They are accessible by the patient and the doctor but not by any other doctors. *Bills* are sent to the patient's family or to her health insurance company, depending on her *PatientPersonalInfo* and *PatientFinancialInfo*. *PatientMedicalInfo* contains general medical information of the patient like allergies. Besides that, the receptionist is able to make *Appointments*, e.g., for a patient's family and the patient's doctor, where the family may sign an *LegalAgreement*, e.g., for special *Prescriptions* or special instructions for the *CarePlan* of the patient. In contrast to [230], permissions for modifying the user set U and the user-to-role assignment relation UA are contained in this model via the virtual objects U_0 and UA_0. The selected RBAC HIS policy does not require a distinct administrative model.

The set of operations OP contains seven operations for accessing the model's objects. While *view* enables a user to read an object; *access* defines further constraints, e.g., the doctor calling this operation must be the patient's doctor [77]. The authors suggest to monitor the usage of *access* and inform the management in case it is used too often.

$$OP = \{\mathit{view}, \mathit{add}, \mathit{modify}, \mathit{access}, \mathit{enter}, \mathit{create}, \mathit{update}, \mathit{sign}\}$$

The model instance contains ten roles. While most of the roles have descriptive names, the roles *MedicalManager* and *ReferredDoctor* need a short explanation. The *MedicalManager* is able to form medical teams (role *MedicalTeam*), consisting of the roles *Doctor* and *Nurse*. The role *ReferredDoctor* indicates a special doctor, to which a patient was referred to by its own doctor, e.g., for obtaining a second opinion.

$$\begin{aligned} R = \{&\mathit{Employee}, \mathit{Manager}, \mathit{Doctor}, \mathit{Nurse}, \mathit{Receptionist}, \mathit{Patient}, \mathit{MedicalManager},\\ &\mathit{MedicalTeam}, \mathit{ReferredDoctor}, \mathit{UserAdmin}\}\end{aligned}$$

The matrix $m : R \times O \to 2^{OP}$ contains the role-to-permission assignment shown in Table B.1. Noteworthy are some policy subtleties that allow for separation of concern. For example, though a user with role *Manager* is allowed to create *RecentMedicalRecords*, only a *Doctor* may add new entries to *RecentMedicalRecords* and view them. Moreover, the roles *MedicalManager* and *ReferredDoctor* are not assigned any permissions; they serve as a kind of attribute in the authorization scheme (see Appendix B.2).

The role hierarchy as introduced in Figure 3.5 is modeled by the relation $RH \subseteq R \times R$, written as $\succeq$, where $\forall r, r' \in R : r \succeq r'$ holds: r inherits the permissions of r'. For example, the role *Doctor* inherits all permissions of the role *Nurse* and has additional permissions reflecting its responsibility and competences. Additionally, [230] discusses some policy characteristics that do not need to be considered in the role hierarchy. For example, the role *ReferredDoctor* will only be assigned to users with the role *Doctor*.

$$\mathit{Manager} \succeq \mathit{MedicalManager}$$
$$\mathit{MedicalManager} \succeq \mathit{Receptionist}$$
$$\mathit{Receptionist} \succeq \mathit{Employee}$$
$$\mathit{Doctor} \succeq \mathit{Nurse}$$
$$\mathit{Nurse} \succeq \mathit{Employee}$$

It remains to define the role exclusion relation $RE \subseteq R \times R$, written as $\approx_{ex}$, where $\forall r, r' \in R : r \approx_{ex} r'$ holds: r and r' are mutually exclusive and may thus not be assigned to the same user via UA.

$$\mathit{Patient} \approx_{ex} \mathit{Employee}$$

Object	Operation	Role	Manager	Doctor	Nurse	Receptionist	Patient	UserAdmin
OldMedicalRecords	view			×			×	
OldMedicalRecords	enter		×					
OldMedicalRecords	access				×			
RecentMedicalRecords	view			×	×		×	
RecentMedicalRecords	enter		×					
RecentMedicalRecords	add			×				
PrivateNotes	view			×			×	
PrivateNotes	add			×				
Prescriptions	view			×				
Prescriptions	modify			×				
PatientMedicalInfo	access		×					
PatientPersonalInfo	access		×					
PatientFinancialInfo	access		×					
CarePlan	view		×		×			
CarePlan	update			×				
Appointment	create					×		
ProgressNotes	add				×			
LegalAgreement	sign						×	
Bills	view						×	
U_o	update							×
ua_o	update							×

Table B.1: Role-to-permission Assignment Matrix

$$
\begin{aligned}
&\mathit{Patient} \approx_{ex} \mathit{ReferredDoctor}\\
&\mathit{Patient} \approx_{ex} \mathit{UserAdmin}\\
&\mathit{Doctor} \approx_{ex} \mathit{Manager}\\
&\mathit{Doctor} \approx_{ex} \mathit{Receptionist}
\end{aligned}
$$

B.2 Authorization Scheme

The policy's authorization scheme consists of state-modifying and non-state-modifying commands. State-modifying commands are:

$$
\begin{aligned}
C_{mod} = \{&\mathit{createUser}, \mathit{destroyUser}, \mathit{assignRole}, \mathit{revokeRole}, \mathit{login}, \mathit{logout}, \mathit{activateRole},\\
&\mathit{deactivateRole}, \mathit{assignReferredDoctorRole}, \mathit{assignPatientRole},\\
&\mathit{assignMedicalTeamRole}, \mathit{revokeEmployeeRole}, \mathit{revokeDoctorRole},\\
&\mathit{revokeReferrefDoctorRole}, \mathit{revokePatientRole}, \mathit{revokeMedicalTeamRole}\}
\end{aligned}
$$

The set of non-state-modifying commands $C_{non-mod} = OP$ is defined by the set of operations OP. Every state-modifying command consists of an antecedent and a consequent. The commands of the RBAC HIS policy are modeled by means of the state transition function δ as follows. Note that $user^{-1} : U \to 2^S$ is the inverted function of $user : S \to U$.

```
$\delta$(q, (createUser $(x_s, x_u)$) ::=
    **if** $(cond_{core}(q, x_s, `U_o`, `update`))$
    **then**
        addUsers $(q, \{x_u\})$
    **end if.**

$\delta$(q, (destroyUser $(x_s, x_u)$) ::=
    **if** $(cond_{core}(q, x_s, `U_o`, `update`))$
    **then**
        deleteUsers(destroySessions $(q, user_q^{-1}(x_u)), \{x_u\})$
    **end if.**

$\delta$(q, (assignRole $(x_s, x_u, x_r)$) ::=
    **if** $cond_{core}(q, x_s, `ua_o`, `update`) \wedge$
        $sod(q, x_u, x_r)$
    **then**
        assignUserToRoles $(q, x_u, \{x_r\})$
    **end if.**

$\delta$(q, (revokeRole $(x_s, x_u, x_r)$) ::=
    **if** $(cond_{core}(q, x_s, `ua_o`, `update`))$
    **then**
        revokeUserFromRoles $(q, x_u, \{x_r\})$
    **end if.**

$\delta$(q, (login $(x_s, x_u)$) ::=
        mapUserSessions(createSessions $(q, \{x_s\}), x_u, \{x_s\})$

$\delta$(q, (logout $(x_s)$) ::=
        destroySessions(unmapUserSessions $(q, user_q(x_s), \{x_s\}), \{x_s\})$

$\delta$(q, (activateRole $(x_s, x_r)$) ::=
        activateRoles $(q, x_s, \{x_r\})$

$\delta$(q, (deactivateRole $(x_s, x_r)$) ::=
        deactivateRoles $(q, x_s, \{x_r\})$

$\delta$(q, (assignReferredDoctorRole $(x_s, x_u)$) ::=
    **if** $cond_{SR}(q, x_s, `Doctor`) \wedge$
        $cond_{UR}(q, x_u, `Doctor`) \wedge$
        $sod(q, x_u, `ReferredDoctor`)$
    **then**
        assignUserToRoles $(q, x_u, \{`ReferredDoctor`\})$
    **end if.**

$\delta$(q, (revokeReferredDoctorRole $(x_s, x_u)$) ::=
    **if** $cond_{SR}(q, x_s, `Doctor`)$
    then
        revokeUserFromRoles $(q, x_u, \{‘\mathit{ReferredDoctor}‘\})$
    end if.

$\delta$(q, (assignPatientRole $(x_s, x_u)$) ::=
    if $\mathit{cond}_{SR}(q, x_s, ‘\mathit{Receptionist}‘)\ \wedge$
        $\mathit{sod}(q, x_u, ‘\mathit{Patient}‘)$
    then
        assignUserToRoles $(q, x_u, \{‘\mathit{Patient}‘\})$
    end if.

$\delta$(q, (revokePatientRole $(x_s, x_u)$) ::=
    if $\mathit{cond}_{SR}(q, x_s, ‘\mathit{Receptionist}‘)$
    then
        revokeUserFromRoles $(q, x_u, \{‘\mathit{Patient}‘\})$
    end if.

$\delta$(q, (assignMedicalTeamRole $(x_s, x_u)$) ::=
    if $\mathit{cond}_{SR}(q, x_s, ‘\mathit{MedicalManager}‘)\ \wedge$
        $(\mathit{cond}_{UR}(q, x_u, ‘\mathit{Doctor}‘)\ \vee\ \mathit{cond}_{UR}(q, x_u, ‘\mathit{Nurse}‘))\ \wedge$
        $\mathit{sod}(q, x_u, ‘\mathit{MedicalTeam}‘)$
    then
        assignUserToRoles $(q, x_u, \{‘\mathit{MedicalTeam}‘\})$
    end if.

$\delta$(q, (revokeMedicalTeamRole $(x_s, x_u)$) ::=
    if $\mathit{cond}_{SR}(q, x_s, ‘\mathit{MedicalManager}‘)$
    then
        revokeUserFromRoles $(q, x_u, \{‘\mathit{MedicalTeam}‘\})$
    end if.
```

Non-state-modifying commands are defined similarly. However, they only consist of an antecedent and are defined by means of the output function λ. For example, the non-state-modifying command *view*() is defined as follows:.

$\lambda(q, (\mathit{view}, (x_s, x_o))) ::= \mathit{cond}_{core}(q, x_s, x_o, \mathit{view})$

C Formal Rule System

This appendix extends the formal rule system defined in Section 4.4.4 by formal rules defined on the algebras *Mapping*[*Domain*, *Target*] (Appendix C.1) and *Matrix*[*Row*, *Column*, *Target*] (Appendix C.2).

C.1 Mapping

In the following, we discuss formal rules defined on the algebra *Mapping*[*Domain*, *Target*]. Since this algebra also models a set of ordered pairs, there are a lot of similarities in the conditions and the derived TCB functions compared to *Relation*. Differences regarding the derived TCB functions only result from the mappings left-total and right-unique properties. Thus, to define formal rules for the algebra *Mapping*, we need to define an additional predicate:

4. $mapping\_sort(x)$: x is of sort *Mapping*

In analogy to *Relation*, the algebra *Mapping* is implemented by a corresponding ADT `Mapping[Domain, Target]` (short `Map_T`), which again is based on the container ADT `Cont_T`, storing tuple elements consisting of type `Dom_T` and `Tar_T`. Consequently, the first rule concerning the constructor of *Mapping* is identical to the rules defined on the constructing operations of set and relation. For this reason, we do not need to add a new rule to the rule system. Only a new interface operator `createMapping()` defined on `Map_T` is required, which encapsulates the constructor operator `createContainer()` of `Cont_T`.

The 2. rule is based on an algebra operation that may be called copy constructor since it unifies an empty `Map_T` object with a given nonempty one. Thus, it is semantically equal to $\cup \in \mathcal{U}_O$. To implement this operation, the interface operation `createMapping()` encapsulating `createContainer()` is required that constructs a new and empty `Map_T` object. The operator `next()` is necessary to iterate the given mapping object and select the next tuple element, which can then be inserted in the newly created object.

The next rule is defined on an algebra operation that inserts a single pair of elements into a given mapping (called *mp_insert*() in Table 4.4). Due to the similarities of the algebra operations *mp_insert*() and *r_insert*(), this rule strongly resembles the second rule of *Relation*. However, the derived function set differs in that it does not contain the ADT operators `equals()` defined on `Tar_T`, `getSecond()`, and `and()`. These functions are not necessary to implement the interface function `mp_insert()` of `Map_T`, since it is sufficient to compare the

2. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$ $\wedge\ \exists! s_{i,A} : mapping\_sort(s_{i,A})$ $\wedge\ mapping\_sort(s_A)$	$\varphi(o_A) :=$ { `Elem_T next(Cont_T c),` `Map_T createMapping(),` `Cont_T insert(Cont_T c, Elem_T e)`} $\beta(\{createContainer, next, insert\}) := \{\cup\}$ $\beta(\{mp\_init\}) := \{\cup\}$ (Interface Function)

domain elements of a tuple due to `Map_T` being a left-total and right-unique relation. The operator `create()` is required to construct the input parameters of `mp_insert()` and are therefore not contained in the mapping β.

3. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$ $\wedge \exists s_{i,A}$: $param\_sort(s_{i,A}, A_{Map})$ $\wedge \exists! s_{j,A} : mapping\_sort(s_{j,A})$ $\wedge\ i \neq j$ $\wedge\ mapping\_sort(s_A)$	$\varphi(o_A) := \{$ `Elem_T nexElement(Cont_T c)`, `Elem_T create(Dom_T d, Tar_T t)`, `Cont_T insert(Cont_T c, Elem_T e)`, `Bool_T equals(Dom_T d`$_1$`, Dom_T d`$_2$`)`, `Dom_T getFirst(Elem_T e)`, `Bool_T false()`, `Bool_T true()`$\}$ $\beta(\{next, insert, equals, getFirst, false, true\}) := \{\cup\}$ $\beta(\{mp\_insert\}) := \{\cup\}$ (Interface Function)

The following rule is in accordance with the 3. rule defined on *Relation*; it is defined on an algebra operation that checks whether a pair of domain and target elements is contained by an mapping object. Since the special characteristics of *Mapping* are irrelevant for this function, the derived set of ADT operators equals the one derived from 3. rule of *Relation*. The only difference lies in the resulting interface functions. Here, an additional interface operator called `mp_isElement()` is added to `Map_T`. Again, `create()` is required to construct the input parameters of this operator.

4. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \in$ $\wedge \exists! s_{i,A} : mapping\_sort(s_{i,A})$ $\wedge \exists s_{j,A}$: $param\_sort(s_{j,A}, A_{Map})$ $\wedge\ i \neq j$	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c)`, `Dom_T create()`, `Bool_T equals(Dom_T d`$_1$`, Dom_T d`$_2$`)`, `Dom_T getFirst(Elem_T e)`, `Tar_T getSecond(Elem_T e)`, `Bool_T false()`, `Bool_T true()`, `Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$ $\beta(\{next, equals, getFirst, getSecond, false, true, and\}) := \{\in\}$ $\beta(\{mp\_isElement\}) := \{\in\}$ (Interface Function)

Since the resulting sets of TCB functions are identical, we can merge both rules into a single one. More precisely, the conditions of both rules are combined as follows; φ remains the same. The disjunctive combination of the individual conditions is necessary, e.g., $(mapping\_sort(s_{i,A}) \vee\ relation\_sort(s_{i,A}))$, since the algebra *Matrix* defines a similar operation $(m\_ \in ())$, but requires other ADT operators. Thus, we must be able to distinguish these algebra operations.

$$\begin{aligned}\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \in\ &\wedge \exists! s_{i,A} : (mapping\_sort(s_{i,A}) \vee\ relation\_sort(s_{i,A}))\\ &\wedge\ \exists s_{j,A} : (param\_sort(s_{j,A}, A_{Map}) \vee param\_sort(s_{j,A}, A_R))\\ &\wedge\ i \neq j\end{aligned}$$

The next two rules resemble the 4. and 5. of *Relation*. The resulting sets of ADT operators slightly differ however, since the underlying operations $mp\_\cup()$ and $mp\_\backslash()$ (Table 4.4) have to consider the algebra property right-uniqueness. Like $mp\_insert()$, $mp\_\cup()$ only inserts a new tuple element if the mapping does not already contain a single pair with the same domain element. To implement this operation, a TCB requires six basic ADT operators to iterate the containers `Cont_T` of the mapping objects, to select the domain elements of the pairs by `getFirst()`, to compare the domain elements by `equals()`, and depending on the comparison result (`true()` or `false()`), to insert the element into the corresponding container. As can be seen, this is a subset of the derived function of the 3. rule. Adding the function `create()` for constructing the input parameter of $mp\_\cup()$ results in the same function set. On top of that, `createMapping()` is required to create an mapping object, in which the newly created elements can be inserted.

Technically speaking, the operation $mp\_\backslash()$ does not have to consider right-uniqueness of the algebra and must compare both tuple elements before deleting a tuple. However, since $mp\_\cup()$ guarantees that an mapping object does not contain two pairs with exactly the same domain element, it is sufficient that $mp\_\backslash()$ only compares the domain elements. Consequently, the necessary TCB function set for $mp\_\backslash()$ is smaller compared to the one derived from $r\_\backslash()$; here, the functions `getSecond`(), `equals`(), and `and`() are not necessary. Besides that `createMapping()` and `mp_insert()` are again required to construct the second input parameter.

5./6. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$ $\wedge\ \forall s_{i,A} : mapping\_sort(s_{i,A})$	$\varphi(o_A) :=$ { `Map_T createMapping()`, `Map_T mp_insert(Map_T mp, Elem_T e)`} $\beta(\{mp\_union\}) := \{\cup\}$ (Interface Function)
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \backslash$ $\wedge\ \forall s_{i,A} : mapping\_sort(s_{i,A})$	$\varphi(o_A) :=$ { `Elem_T next(Cont_T c)`, `Map_T createMapping()`, `Map_T mp_insert(Map_T mp, Elem_T e)`, `Bool_T equals(Dom_T d`$_1$`, Dom_T d`$_2$`)`, `Cont_T delete(Cont_T c, Elem_T e)`, `Domain_T getFirst(Elem_T e)`, `Bool_T false()`, `Bool_T true()`, $\beta(\{next, equals, delete, getFirst, false, true\}) := \{\backslash\}$ $\beta(\{mp\_diff\}) := \{\backslash\}$ (Interface Function)

Besides their conditions, the following rules are similar to the 6. and 7. rule of *Relation*. The reason is that the algebra operations, on which the rules are defined, are semantically equal such that they require exactly the same TCB functions for their implementation. Hence, we can subsume these rules to two rules; only the interface operators are defined for different ADTs: While `r_domainSub()` and `mp_rangeSub()` belong to `Rel_T`, `Map_T` contains `mp_domainSub()` and `mp_rangeSub()`. The derived function set of the 6. rule equals the function set derived from rule 5 and the first two rules of *Set* so that these rules can be reused. While `r_delete()` is required to implement the algebra operations, `createSet()` and `s_insert()` are required to construct the second input parameter.

The function set of rule 7. then differs in that it requires the operator `getSecond()` instead of `getFirst()`. This results in that no rules defined on *Mapping* can be reused; only the first two rules of *Set* can be reused to derive functions to construct the second input parameter.

7./8. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = ⩤$ $\vee\, \alpha(o_A) = \lhd)$ $\wedge\, \exists!\, s_{i,A} : \mathit{mapping\_sort}(s_{i,A})$ $\wedge\, \mathit{mapping\_sort}(s_A)$	$\varphi(o_A) := \{$ `Map_T mp_delete(Map_T mp, Eleme_T e),` `Set_T createSet(),` `Set_T s_insert(Set_T c,Elem_T e)`$\}$ $\beta(\{\mathit{next}, \mathit{equals}, \mathit{delete}, \mathit{getFirst}, \mathit{false}, \mathit{true}\}) := \{⩤, \lhd\}$ $\beta(\{\mathit{mp\_domainSub}\}) := \{⩤\}$ (Interface Function) $\beta(\{\mathit{mp\_domainRes}\}) := \{\lhd\}$ (Interface Function)
$\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = ⩥$ $\vee\, \alpha(o_A) = \rhd)$ $\wedge\, \exists!\, s_{i,A} : \mathit{mapping\_sort}(s_{i,A})$ $\wedge\, \mathit{mapping\_sort}(s_A)$	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),` `Set_T createSet(),` `Set_T s_insert(Set_T c,Elem_T e)` `Bool_T equals(Dom_T d`$_1$`,Dom_T d`$_2$`),` `Cont_T delete(Con_T c, Elem_T e),` `Tar_T getSecond(Elem_T e),` `Bool_T false(),` `Bool_T true()`$\}$ $\beta(\{\mathit{next}, \mathit{equals}, \mathit{delete}, \mathit{getSecond}, \mathit{false}, \mathit{true}\}) := \{⩥, \rhd\}$ $\beta(\{\mathit{r\_rangeSub}\}) := \{⩥\}$ (Interface Function) $\beta(\{\mathit{r\_rangeRes}\}) := \{\rhd\}$ (Interface Function)

Result are two rules, whose subsumed conditions are defined as follows:

7. $\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = ⩤ \,\vee\, \alpha(o_A) = \lhd)$
   $\wedge\; \exists!\, s_{i,A} : (\mathit{mapping\_sort}(s_{i,A}) \vee \mathit{relation\_sort}(s_{i,A}))$
   $\wedge\; (\mathit{mapping\_sort}(s_A) \vee \mathit{relation\_sort}(s_A))$
8. $\exists o_A \in \mathcal{O}_A : \alpha(o_A) = ⩥ \,\vee\, \alpha(o_A) = \rhd$
   $\wedge\; \exists!\, s_{i,A} : (\mathit{mapping\_sort}(s_{i,A}) \vee \mathit{relation\_sort}(s_{i,A}))$
   $\wedge\; (\mathit{mapping\_sort}(s_A) \vee \mathit{relation\_sort}(s_A))$

In contrast to the 4. rule, which we have also combined with the corresponding rule of *Relation*, the subsumed conditions must not necessarily distinguish the sorts of the input and output parameters. The reason is that the algebra foundation does not define an additional algebra with similar operations. In case, the algebra foundation were modified to do so and the implementation of the additional algebra operations required different ADT operators than shown here, it is important to distinguish the input and output parameter sorts. To then avoid incomplete conditions, the redundancy in these conditions is maintained.

The remaining rules are defined on *Mapping*-specific operations and thus bear no resemblance to any rules defined on *Relation*. Rule 8 is based on an algebra operation that selects a target element belonging to a given domain element. In other words, it returns the functional value of a given element of the mapping's domain set. In order to implement this operation, a TCB requires eight functions, encapsulated by the interface operator `mp_getMapping()`. On top of that, the operator `create()` is necessary to create the second input parameter of the interface operator. Since `mp_getMapping()` returns an object of type `Set_T`, the function `createSet()` is required to create an object, in which the selected target element has to be inserted. If no target element is found, an empty set will be returned. Note that for inserting an element in the set, the TCB does not require all functions that implement `s_insert()`.

The reason is as follows: Due to the mapping being a right-unique relation, a domain element cannot have more than one target elements. Thus, in any case the return set will have either one or null elements; element redundancy cannot happen. Consequently, all functions that contribute to avoiding redundancy (2./4. rule of *Set*) are not required; only the operator `insert()` defined on `Cont_T` must be implemented.

9. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = \mapsto$
 $\wedge\ \exists!\, s_{i,A} : \mathit{mapping\_sort}(s_{i,A})$
 $\wedge\ \exists!\, s_{i,A} :$
 $\mathit{param\_sort}(s_{i,A}, A_{Map})$
 $\wedge\ \mathit{set\_sort}(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	 `Set_T createSet(),`
	 `Dom_T create(),`
	 `Cont_T insert(Cont_T c, Elem_T e),`
	 `Bool_T equals(Dom_T d`$_1$`,Dom_T d`$_2$`),`
	 `Tar_T getFirst(Elem_T e),`
	 `Tar_T getSecond(Elem_T e),`
	 `Bool_T false(), Bool_T true()`$\}$
	 $\beta(\{\mathit{next}, \mathit{createContainer}, \mathit{insert}, \mathit{equals}, \mathit{getFirst}, \mathit{getSecond}, \mathit{false}, \mathit{true}\}) := \{\mapsto\}$
	 $\beta(\{\mathit{mp\_getMapping}\}) := \{\mapsto\}$ (Interface Function)

The last rule for *Mapping* derives TCB functions for an algebra operation that is generally known as functional overriding. Even though it is not exactly necessary, the rule's condition also considers the sorts of the operation's input and output parameters. The reason is the same as mentioned earlier (subsumed conditions for the 6./7. rule): currently, the algebra foundation does not specify an algebra that contains an operation with the same semantics. However, if this happened and the operation required different TCB functions, the sorts of the operations' input and output parameters are the only way to differ them. The derived functions equal the ones derived from rule 4 and 5. While `mp_insert()` and `mp_delete()` implement the algebra operation `createMapping()` and is necessary to construct the input parameter. Note that `mp_insert()` is also required for the input parameter.

10. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : (\alpha(o_A) = \oplus$ $\wedge\ \exists\, s_{i,A} : \mathit{mapping\_sort}(s_{i,A})$ $\wedge\ \mathit{mapping\_sort}(s_A)$	$\varphi(o_A) := \{$ `Map_T createMapping(),` `Map_T mp_insert(Map_T mp, Elem_T e),` `Map_T delete(Map_T mp, Elem_T e)`$\}$ $\beta(\{\mathit{next}, \mathit{insert}, \mathit{delete}, \mathit{equals}, \mathit{getFirst}, \mathit{false}, \mathit{true}\}) := \{\oplus\}$ $\beta(\{\mathit{mp\_override}\}) := \{\oplus\}$ (Interface Function)

Having derived the TCB functions for *Mapping* based on these rules, the result are eight ADTs as already known from *Relation*. Due to the ADTs' great resemblance, we refrain from presenting them again. The only difference is the ADT `Map_T`, which now replaces `Rel_T` and is presented in Table C.1. As with `Rel_T`, the semantics of the ADT's operators is defined by the mapping β such that we do not need to provide an additional set of axioms.

sorts	`Map_T`
import	`Bool_T`, `Dom_T`, `Tar_T`, `Set_T`
operators	`createMapping:` $\rightarrow$ `Map_T`
	`mp_insert:` `Map_T` $\times$ `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Map_T`
	`mp_isElement:` `Map_T` $\times$ `Dom_T` $\times$ `Tar_T` $\rightarrow$ `Bool_T`
	`mp_union:` `Map_T` $\times$ `Map_T` $\rightarrow$ `Map_T`
	`mp_diff:` `Map_T` $\times$ `Map_T` $\rightarrow$ `Map_T`
	`mp_domainSub:` `Map_T` $\times$ `Set_T` $\rightarrow$ `Map_T`
	`mp_rangeSub:` `Map_T` $\times$ `Set_T` $\rightarrow$ `Map_T`
	`mp_domainRes:` `Map_T` $\times$ `Set_T` $\rightarrow$ `Map_T`
	`mp_rangeRes:` `Map_T` $\times$ `Set_T` $\rightarrow$ `Map_T`
	`mp_getMapping:` `Map_T` $\times$ `Dom_T` $\rightarrow$ `Set_T`
	`mp_override:` `Map_T` $\times$ `Map_T` $\rightarrow$ `Map_T`

Table C.1: ADT `Mapping[Domain,Target]` (`Map_T`)

C.2 Matrix

The remaining rules of the formal rule system are concerned with the algebra *Matrix*[*Row*, *Column*, *Target*], modeling a set of triples, each of which representing a matrix cell. Due to its great resemblance to the algebra *Mapping* (both model left-total and right-unique relations), there are many similarities in the conditions and the derived TCB functions. Differences in the latter only result from dealing with triple elements.

To distinguish the operations of *Matrix* from operations of *Mapping*, we need an extra predicate. Additionally, to distinguish *Matrix*-specific operations that require different TCB functions, we need two additional predicates (6. and 7.) that differ between the parameter sorts of the domain elements of *Matrix*,

5. *matrix_sort*(x): x is of sort *Matrix*
6. *row_sort*(x): x is of sort *Row*
7. *column_sort*(x): x is of sort *Column*

Just like the other algebras, *Matrix* produces a corresponding ADT `Matrix_T` that provides all interface operators as defined by its algebra. `Matrix_T` is based on an container ADT `Cont_T`, which stores triple elements `Elem_T` of type `Row_T`, `Col_T`, and `Set_T`. The third element is specified by a set of target elements `Tar_T`, where `Set_T` is implemented by the container data type `Cont_T` as already discussed. Here, we have the same conflict: `Cont_T` being a container data type for different element types. We analogously solve this problem by deriving the triple element type `TrElem_T` from the abstract element `Elem_T`. Compared to `Rel_T` and `Map_T`, we now have three basic types `Row_T`, `Col_T`, and `Tar_T` due to type safety. Again, they may collapse into two types or even one, depending on the definition of a model-specific matrix, e.g., $m : Subjects \times Subjects \rightarrow 2^{Rights}$, and the specific platform implementation where all types may be represented by a primitive string or integer type.

The first rule concerning the constructor of *Matrix* is identical to the rules defined on the constructing operations of the previous algebras. The only difference is that the derived TCB function `createContainer()` specified on `Cont_T` is now encapsulated by a new interface function called `createMatrix()` contained in the operator set of ADT `Matrix_T`.

The 2. rule is based on an algebra operation that may be called copy constructor since it unifies an empty `Matrix_T` object with a given nonempty one. Thus, it is semantically equal to $\cup \in \mathcal{U}_O$. To implement this operation, the interface operation `createMatrix()` encapsulating `createContainer()` is required that constructs a new and empty `Matrix_T` object. The operator `next()` is necessary to iterate the given matrix and select the next triple element, which can then be inserted in the newly created object.

2. Condition	**TCB Functions and Mapping to Universe Operations**
$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$ $\wedge\ \exists! s_{i,A} : \mathit{matrix\_sort}(s_{i,A}, A)$ $\wedge\ \mathit{matrix\_sort}(s_A)$	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),` `Matrix_T createMatrix),` `Cont_T insert(Cont_T c, Elem_T e)`$\}$ $\beta(\{\mathit{createContainer}, \mathit{next}, \mathit{insert}\}) := \{\cup\}$ $\beta(\{\mathit{m\_init}\}) := \{\cup\}$ (Interface Function)

The 3. rule bears great resemblance to the 2. rule of *Mapping*; both are defined on algebra operations that insert a single tuple element while guaranteeing the left-total and right-unique properties of their algebra. However, here we do not add an entire cell, but insert only a new target element into the set of target elements that defines the cell content. Thus, it is now necessary to check the uniqueness of a pair of domain elements (`Row_T` and `Col_T`). For this purpose, the operator `equals()` as well as a logical operator `and()` of `Bool_T` are additionally required. In order to actually insert an a target element in a set, we can refer to the interface function `s_insert()` and the basic ADT operators that implement it (2. rule of `Set_T`). A triple element contained in the matrix object then is replaced by a new one by deleting the it and inserting the new triple. Besides the function `create()`, which is necessary to construct the input parameters of type `Row_T`, `Col_T`, and `Tar_T`, all other functions are used by the interface function `m_insert()` of `Matrix_T`.

3. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$
 $\wedge\ \exists s_{i,A} : \mathit{param\_sort}(s_{i,A}, A)$
 $\wedge\ \exists! s_{j,A} : \mathit{matrix\_sort}(s_{j,A})$
 $\wedge\, i \neq j$
 $\wedge\ \mathit{matrix\_sort}(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	 `Elem_T create(Row_T r, Col_T c,`
	 `Set_T s),`
	 `Set_T createSet(),`
	 `Cont_T delete(Cont_T c, Elem_T e),`
	 `Cont_T insert(Cont_T c, Elem_T e),`
	 `Set_T s_insert(Set_T c, Elem_T e),`
	 `Bool_T equals(Row_T r`$_1$`, Row_T r`$_2$`),`
	 `Row_T getFirst(Elem_T e),`
	 `Col_T getSecond(Elem_T e),`
	 `Set_T getThird(Elem_T e),`
	 `Bool_T false(),`
	 `Bool_T true(),`
	 `Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	 $\beta(\{\mathit{next}, \mathit{create}, \mathit{insert}, \mathit{delete}, \mathit{equals}, \mathit{getFirst}, \mathit{getSecond},$
	 $\mathit{getThird}, \mathit{false}, \mathit{true}, \mathit{and}, \mathit{createContainer}, \}) := \{\cup\}$
	 $\beta(\{\mathit{m\_insert}\}) := \{\cup\}$ (Interface Function)

The 4. rule of *Matrix* is similar to the 3. rule of *Relation* and *Mapping*. The main difference is that in contrast to the latter where the existence of a tuple element in a container is checked,

this operation checks the existence of a specific target element in the set of target elements of a matrix cell. Consequently, the derived TCB function sets cannot be equal such that we cannot subsume these rules into one. Additional TCB functions are `getThird()` to select the target set of the given cell, `next()` to iterate the container `Cont_T` of the target set, and `equals()`. All operators that are needed to test the existence of a set element are encapsulated by the interface function `s_isElement()` of `Set_T` (3. rule of *Set*). Besides the constructor operator for the basic elements (`create()`), the remaining functions as well as the constructors of `Bool_T` objects implement the interface function `m_isElement()`.

4. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \in$
$\wedge\ \exists! s_{i,A} : \mathit{matrix\_sort}(s_{i,A})$
$\wedge\ \exists s_{j,A} : \mathit{param\_sort}(s_{j,A}, A_M)$
$\wedge\ i \neq j$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	`Bool_T s_isElement(Set_T s, Tar_T t),`
	`Row_T create(),`
	`Bool_T equals(Row_T r`$_1$`, Row_T r`$_2$`),`
	`Row_T getFirst(Elem_T e),`
	`Col_T getSecond(Elem_T e),`
	`Tar_T getThird(Elem_T e),`
	`Bool_T false(),`
	`Bool_T true(),`
	`Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	$\beta(\{\mathit{next}, \mathit{equals}, \mathit{getFirst}, \mathit{getSecond}, \mathit{getThird}, \mathit{false}, \mathit{true}, \mathit{and}\}) := \{\in\}$
	$\beta(\{\mathit{m\_isElement}\}) := \{\in\}$ (Interface Function)

The following rule is similar to the 8. rule of *Mapping*, which is based on an operation that selects the functional value of a given domain element. Again, the derived TCB function set is larger due the triple elements. In case a given cell does not exist, an empty set is returned, which first has to be created by `createSet()`. All other basic functions are directly called by the interface function `m_cellContent()` defined on `Matrix_T`.

5. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \mapsto$
$\wedge\ \exists! s_{i,A} : \mathit{matrix\_sort}(s_{i,A})$
$\wedge\ \exists s_{j,A} : \mathit{param\_sort}(s_{j,A}, A_M)$
$\wedge i \neq j$
$\wedge\ \mathit{set\_sort}(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(MCont_T c),`
	`Set_T createSet(),`
	`Row_T create(),`
	`Bool_T equals(Row_T r`$_1$`, Row_T r`$_2$`),`
	`Row_T getFirst(Elem_T e),`
	`Col_T getSecond(Elem_T e),`
	`Tar_T getThird(Elem_T e),`
	`Bool_T false(),`
	`Bool_T true(),`
	`Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	$\beta(\{\mathit{next}, \mathit{equals}, \mathit{getFirst}, \mathit{getSecond}, \mathit{getThird}, \mathit{false}, \mathit{true}, \mathit{and}, \mathit{createContainer}\}) := \{\mapsto\}$
	$\beta(\{\mathit{m\_cellContent}\}) := \{\mapsto\}$ (Interface Function)

The following rules are defined on *Matrix*-specific operations that modify the cells of a matrix. The 6. rule is based on an operation that deletes single elements from the target set contained in a given cell. To support this algebra operation, the container `Cont_T` implementing `Matrix_T` has to be iterated by `next()` returning a triple, whose row and column elements have to be compared to the appropriate input parameters. Once the correct cell has been found, the inputed target element can be deleted from the target set of the triple by using the interface function `s_diff()` of `Set_T`. Thus, the old triple must be removed from the matrix to be replaced by the modified one. Except for the function `create()`, all functions are called by the interface function `m_delete()` of `Matrix_T`.

6. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \setminus$
$\wedge\ \exists! s_{i,A} : matrix\_sort(s_{i,A})$
$\wedge\ \exists s_{j,A} : param\_sort(s_{j,A}, A_M)$
$\wedge\ i \neq j$
$\wedge\ matrix\_sort(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	`Set_T s_diff(Set_T c, Elem_T t),`
	`Elem_T create(Row_T r, Col_T c,`
	`Set_T s),`
	`Cont_T delete(Cont_T c, Elem_T e),`
	`Cont_T insert(Cont_T c, Elem_T e),`
	`Bool_T equals(Row_T r`$_1$`, Row_T r`$_2$`),`
	`Bool_T equals(Col_T c`$_1$`, Col_T c`$_2$`),`
	`Row_T getFirst(Elem_T e),`
	`Col_T getSecond(Elem_T e),`
	`Set_T getThird(Tlem_T e),`
	`Bool_T false(),`
	`Bool_T true(),`
	`Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$
	$\beta(\{next, equals, insert, delete, getFirst, getSecond, getThird,$ $false, true, and\}) := \{\setminus\}$
	$\beta(\{m\_delete\}) := \{\setminus\}$ (Interface Function)

Rules 7 is defined on operations that add either a new row or a new column to a matrix. In order to support these functions, an additional data structure is required that stores all existing columns (respectively rows) so that new cells can be created by combining the inputed row with the existing columns. Here, we also apply the ADT `Set_T` since is must already be present in a TCB to support the algebra *Matrix* itself so that we do not have to add new TCB functions. Before a new cell is inserted into the matrix, all existing cells are iterated by `next()` to check whether the new cell already exists. Only if it does not, a new triple element is created and inserted into the matrix. As can be seen, already for creating the triple element, the functions `createSet()` and `s_insert()` are required and can thus be reused for implementing the required set of columns/rows. `create()` is also required to create the second input parameter of the interface functions `m_addRow()` and `m_addColumn()`.

7. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \cup$
$\wedge\ \exists! s_{i,A} : (row\_sort(s_{i,A})$
$\vee\ column\_sort(s_{i,A}))$
$\wedge\ \exists! s_{j,A} : matrix\_sort(s_{j,A})$
$\wedge\ i \neq j$
$\wedge\ matrix\_sort(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	`Elem_T create(Row_T r, Col_T c,`
	`Set_T s),`
	`Set_T createSet(),`
	`Cont_T insert(Cont_T c, Elem_T e),`
	`Set_T s_insert(Set_T c, Col_T c),`
	`Bool_T equals(Col_T c`$_1$`, Col_T c`$_2$`),`
	`Row_T getFirst(Elem_T e),`
	`Col_T getSecond(Elem_T e),`
	`Bool_T false(),`
	`Bool_T true(),`
	`Bool_T and(Bool_T b`$_1$`,Bool_T b`$_2$`)`$\}$

	$\beta(\{next, create, createContainer, insert, equals, getFirst, getSecond, false, true, and\}) := \{\cup\}$ $\beta(\{m\_addRow\}) := \{\cup\}$ (Interface Function) $\beta(\{m\_addColumn\}) := \{\cup\}$ (Interface Function)

The remaining rule is defined on algebra operations that delete a single row or column from a matrix. Even though these algebra functions are contrary to the functions dealt with by rule 7, the derived TCB function set differs only slightly. First, rows and columns naturally are deleted (`delete()` instead of `insert()`). Second, `createSet()` and `s_insert()` are not required, since the collections of existing rows or columns are not necessary here. Moreover, `and()` and `equals()` defined on `Col_T` are not required since we only have to compare the row elements of the matrix cells and not the entire domain elements. In contrast, `getThird()` is necessary to construct the matrix element to be deleted.

8. Condition	**TCB Functions and Mapping to Universe Operations**

$\exists o_A \in \mathcal{O}_A : \alpha(o_A) = \setminus$
$\wedge \exists! s_{i,A} : (row\_sort(s_{i,A})$
$\vee\, column\_sort(s_{i,A}))$
$\wedge \exists! s_{j,A} : matrix\_sort(s_{j,A})$
$\wedge\, i \neq j$
$\wedge\, matrix\_sort(s_A)$
	$\varphi(o_A) := \{$ `Elem_T next(Cont_T c),`
	`Row_T create(),`
	`Cont_T delete(Cont_T c, Elem_T e),`
	`Bool_T equals(Row_T r`$_1$`, Row_T r`$_2$`),`
	`Row_T getFirst(Elem_T e),`
	`Row_T getSecond(Elem_T e),`
	`Col_T getThird(Elem_T e),`
	`Bool_T false(),`
	`Bool_T true()`$\}$
	$\beta(\{next, create, delete, equals, getFirst, getSecond, getThird, false, true\}) := \{\setminus\}$
	$\beta(\{m\_deleteRow\}) := \{\setminus\}$ (Interface Function)
	$\beta(\{m\_deleteColumn\}) := \{\setminus\}$ (Interface Function)

Applying these rules on an algebra that contains all of the mentioned operations, the resulting ADT `Matrix_T` is specified as shown by Table C.2. As can be seen, it imports all other ADTs besides `Cont_T` to define its operators. The remaining ADTs are similar to the ones already explained in the previous paragraphs; for this reason we do not discuss them again.

sorts	`Matrix_T`
import	`Bool_T`, `Row_T`, `Col_T`, `Tar_T`, `Set_T`

operators	`createMatrix:` $\rightarrow$ `Matrix_T`
	`m_init: Matrix_T` $\rightarrow$ `Matrix_T`
	`m_insert: Matrix_T` $\times$ `Row_T` $\times$ `Col_T` $\times$ `Tar_T` $\rightarrow$ `Matrix_T`
	`m_isElement: Matrix_T` $\times$ `Row_T` $\times$ `Col_T` $\times$ `Tar_T` $\rightarrow$ `Bool_T`
	`m_cellContent: Matrix_T` $\times$ `Row_T` $\times$ `Col_T` $\rightarrow$ `Set_T`
	`m_delete: Matrix_T` $\times$ `Row_T` $\times$ `Col_T` $\times$ `Tar_T` $\rightarrow$ `Matrix_T`
	`m_addRow: Matrix_T` $\times$ `Row_T` $\rightarrow$ `Matrix_T`
	`m_addColumn: Matrix_T` $\times$ `Col_T` $\rightarrow$ `Matrix_T`
	`m_deleteRow: Matrix_T` $\times$ `Row _T` $\rightarrow$ `Matrix_T`
	`m_deleteColumn: Matrix_T` $\times$ `Col_T` $\rightarrow$ `Matrix_T`

Table C.2: ADT `Matrix[Row,Column,Target]` (`Matrix_T`)

Instead we focus on the ADTs' interrelations illustrated in Figure C.1. As can be seen, a new element type `TrElem_T` is derived from the abstract class `Elem_T`, implementing the shared functions `create()` and `equals()`. Additionally, it provides the triple-specific functions `getFirst()`, `getSecond()`, and `getThird()`. The triple contains three elements of type `Row_T`, `Col_T`, and `Set_T` where the latter is a set of `Tar_T`. Besides an object of type `Cont_T`, `Matrix_T` contains two additional attributes of type `Set_T` that are necessary to support the *Matrix*-specific operations `m_addRow()` and `m_addColumn()`. The latter respectively require a separate collection of the already existing matrix rows and columns. Here, we have reused the type `Set_T` leading to an additional composition of `Matrix_T` and `Set_T` and associations between `Cont_T`, `Row_T`, and `Col_T`.

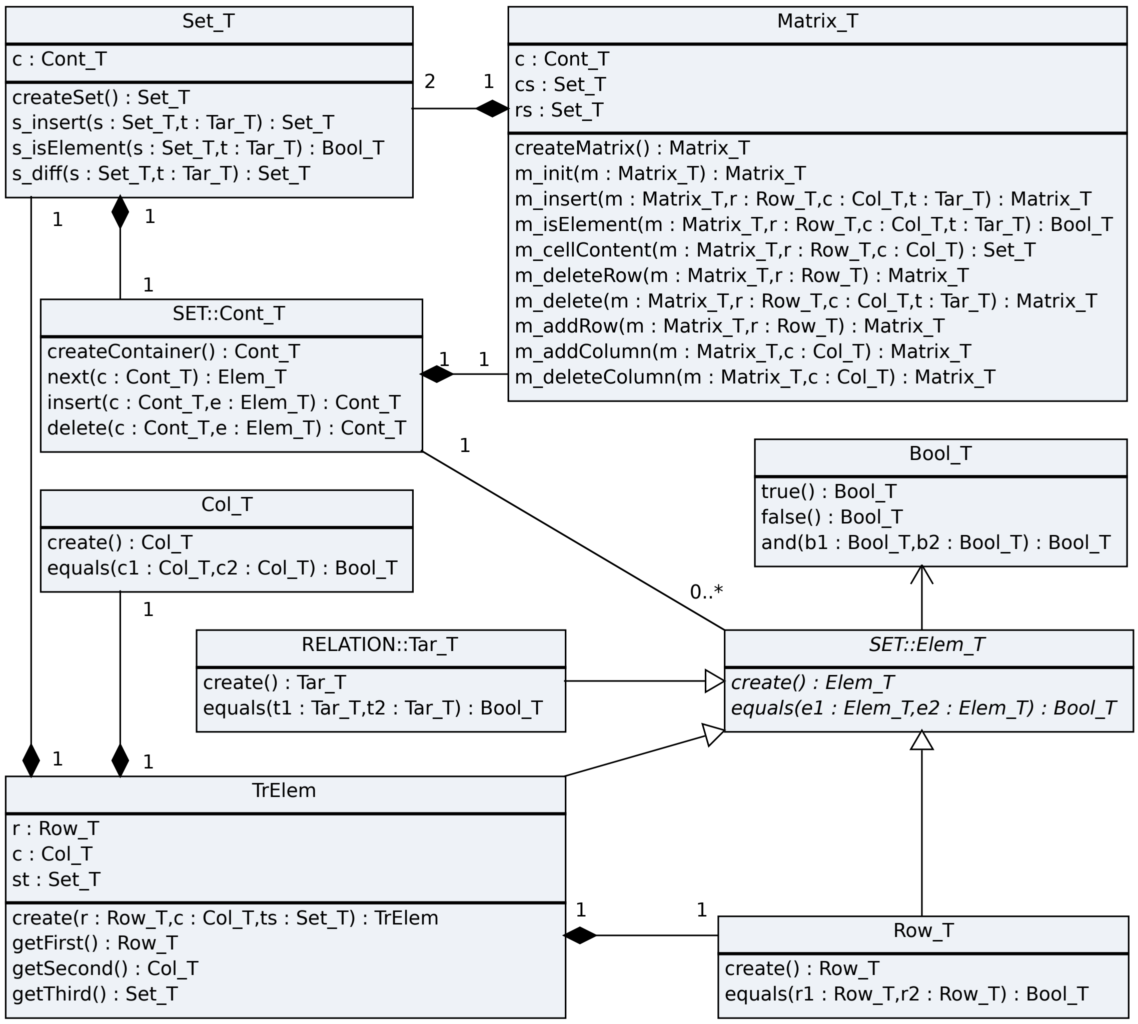
Set_T
c : Cont_T
createSet() : Set_T
s_insert(s : Set_T,t : Tar_T) : Set_T
s_isElement(s : Set_T,t : Tar_T) : Bool_T
s_diff(s : Set_T,t : Tar_T) : Set_T
Matrix_T
c : Cont_T
cs : Set_T
rs : Set_T
createMatrix() : Matrix_T
m_init(m : Matrix_T) : Matrix_T
m_insert(m : Matrix_T,r : Row_T,c : Col_T,t : Tar_T) : Matrix_T
m_isElement(m : Matrix_T,r : Row_T,c : Col_T,t : Tar_T) : Bool_T
m_cellContent(m : Matrix_T,r : Row_T,c : Col_T) : Set_T
m_deleteRow(m : Matrix_T,r : Row_T) : Matrix_T
m_delete(m : Matrix_T,r : Row_T,c : Col_T,t : Tar_T) : Matrix_T
m_addRow(m : Matrix_T,r : Row_T) : Matrix_T
m_addColumn(m : Matrix_T,c : Col_T) : Matrix_T
m_deleteColumn(m : Matrix_T,c : Col_T) : Matrix_T
SET::Cont_T
createContainer() : Cont_T
next(c : Cont_T) : Elem_T
insert(c : Cont_T,e : Elem_T) : Cont_T
delete(c : Cont_T,e : Elem_T) : Cont_T
Col_T
create() : Col_T
equals(c1 : Col_T,c2 : Col_T) : Bool_T
Bool_T
true() : Bool_T
false() : Bool_T
and(b1 : Bool_T,b2 : Bool_T) : Bool_T
RELATION::Tar_T
create() : Tar_T
equals(t1 : Tar_T,t2 : Tar_T) : Bool_T
SET::Elem_T
create() : Elem_T
equals(e1 : Elem_T,e2 : Elem_T) : Bool_T
TrElem
r : Row_T
c : Col_T
st : Set_T
create(r : Row_T,c : Col_T,ts : Set_T) : TrElem
getFirst() : Row_T
getSecond() : Col_T
getThird() : Set_T
Row_T
create() : Row_T
equals(r1 : Row_T,r2 : Row_T) : Bool_T
2
1
1
1
1
1
1
1
0..*
1
1
1
1
1

Figure C.1: ADTs Derived from $Matrix[Row, Column, Target]$

D TCB Specification for the RBAC HIS Policy

This appendix contains the complete TCB specification in Event-B for the core-based RBAC model discussed in Section 3.4.5 and the RBAC HIS policy formalized by this model in Appendix B. We have created this specification by applying the specification method introduced in Section 5.4. A detailed discussion of this TCB specification along with the applied proof techniques is given in [178].

D.1 Context `rbac_static`

```
CONTEXT rbac_static
SETS
  USER
  ROLE
  SESSION
  OBJECT
  OPERATION
CONSTANTS
  M      > role-to-permission assignment matrix
  RH     > role hierarchy
  RE     > role exclusion relation
AXIOMS
  M_t:  M ∈ ROLE × OBJECT ⇸ ℙ(OPERATION)
  M1:  ∀r, o·r ∈ ROLE ∧ o ∈ OBJECT ∧ (r ↦ o ∈ dom(M))
       ⇒ finite(M(r ↦ o))        > finiteness

  RH_t:  RH ∈ ROLE ↔ ROLE
  RH1:  RH = RH; RH        > transitivity and reflexivity
  RH2:  ∀r1, r2·
            r1 ∈ ROLE ∧ r2 ∈ ROLE ∧
            (r1 ↦ r2) ∈ RH ∧ (r2 ↦ r1) ∈ RH
       ⇒ r1 = r2       > antisymmetry

  RE_t:  RE ∈ ROLE ↔ ROLE
  RE1:  RE = RE⁻¹       > symmetry
  RE2: ∀r1,r2·
           r1 ∈ ROLE ∧ r2 ∈ ROLE ∧
           (r1 ↦ r2) ∈ RE
      ⇒ r1 ≠ r2       >  irreflexivity
END
```

D.2 Context rbac_state

```
CONTEXT rbac_state
EXTENDS
  rbac_static
CONSTANTS
  STATE      >  state space
  CONSISTENT_STATE
  U_q      >  projection function
  S_q
  UA_q
  user_q
  roles_q
AXIOMS
  st: STATE = {
           U ↦ S ↦ UA ↦ user ↦ roles |
           U ∈ ℙ(USER) ∧ S ∈ ℙ(SESSION) ∧
           UA ∈ USER ↔ ROLE ∧ dom(UA) ⊆ U ∧
           user ∈ SESSION ⇸ USER ∧ dom(user) ⊆ S ∧ ran(user) ⊆ U ∧
           roles ∈ SESSION ⇸ ℙ(ROLE) ∧ dom(roles) = S ∧
           ( ∀s · s ∈ S ⇒ finite(roles(s)) ) ∧
           ( ∀s · s ∈ dom(user) ⇒ roles(s) ⊆ UA[ {user(s)} ] )
       }
  cst: CONSISTENT_STATE = {
           U ↦ S ↦ UA ↦ user ↦ roles |
           U ↦ S ↦ UA ↦ user ↦ roles ∈ STATE ∧
           dom(user) = S
       }
  cst1: CONSISTENT_STATE ⊆ STATE      theorem

  st-i: ∀U,S,UA,user,roles · (
           ∃q · q ∈ STATE ∧ q = U ↦ S ↦ UA ↦ user ↦ roles
       ) ⇒
           dom(UA) ⊆ U ∧
           dom(user) ⊆ S ∧ ran(user) ⊆ U ∧
           dom(roles) = S ∧ ( ∀s · s ∈ S ⇒ finite(roles(s)) ) ∧
           ( ∀s · s ∈ dom(user) ⇒ roles(s) ⊆ UA[ {user(s)} ] )      theorem      > mapping
       invariants on variables
```

cst-i: $\forall U, S, UA, user, roles \cdot ($
$\exists q \cdot q \in CONSISTENT\_STATE \land$
$q = U \mapsto S \mapsto UA \mapsto user \mapsto roles$
$) \Rightarrow$
$dom(UA) \subseteq U \land$
$dom(user) = S \land ran(user) \subseteq U \land$
$dom(roles) = S \land (\forall s \cdot s \in S \Rightarrow finite(roles(s))) \land$
$(\forall s \cdot s \in dom(user) \Rightarrow roles(s) \subseteq UA[\{user(s)\}])$ **theorem**

st1: $\forall q \cdot q \in STATE \Rightarrow$
$prj2(prj1(q)) \in SESSION \rightarrowtail USER \land$
$prj1(prj1(q)) \in \mathbb{P}(USER) \times \mathbb{P}(SESSION) \times (USER \leftrightarrow ROLE)$ **theorem** > only for ease of proof

st2: $\forall q \cdot q \in STATE \Rightarrow$
$prj2(q) \in SESSION \rightarrowtail \mathbb{P}(ROLE) \land$
$prj1(q) \in \mathbb{P}(USER) \times \mathbb{P}(SESSION) \times (USER \leftrightarrow ROLE) \times$
$(SESSION \leftrightarrow USER)$ **theorem**

stU-t: $U\_q \in STATE \rightarrow \mathbb{P}(USER)$

stS-t: $S\_q \in STATE \rightarrow \mathbb{P}(SESSION)$

stA-t: $UA\_q \in STATE \rightarrow (USER \leftrightarrow ROLE)$

stu-t: $user\_q \in STATE \rightarrow (SESSION \rightarrowtail USER)$

str-t: $roles\_q \in STATE \rightarrow (SESSION \rightarrowtail \mathbb{P}(ROLE))$

st-d: $\forall q \cdot q \in STATE \Rightarrow U\_q(q) = prj1(prj1(prj1(prj1(q)))) \land$
$\forall q \cdot q \in STATE \Rightarrow S\_q(q) = prj2(prj1(prj1(prj1(q)))) \land$
$\forall q \cdot q \in STATE \Rightarrow UA\_q(q) = prj2(prj1(prj1(q))) \land$
$\forall q \cdot q \in STATE \Rightarrow user\_q(q) = prj2(prj1(q)) \land$
$\forall q \cdot q \in STATE \Rightarrow roles\_q(q) = prj2(q)$

st3: $\forall q \cdot q \in STATE \Rightarrow ($
$\exists U, S, UA, user, roles \cdot$
$U \mapsto S \mapsto UA \mapsto user \mapsto roles = q \land$
$U\_q(q) = U \land$
$S\_q(q) = S \land$
$UA\_q(q) = UA \land$
$user\_q(q) = user \land$
$roles\_q(q) = roles$
$)$ **theorem** > mapping state components to variables

END

D.3 Context `rbac_userhandling`

```
CONTEXT rbac_userhandling
EXTENDS
  rbac_state
CONSTANTS
  addUsers
  deleteUsers
AXIOMS
  add_t: addUsers ∈ STATE × ℙ(USER) → STATE
  add_d: ∀u, q ·
            q ∈ STATE ∧
            u ∈ ℙ(USER) ∧ u ∩ U_q(q) = ∅
      ⇒ addUsers(q ↦ u) = (U_q(q) ∪ u) ↦ S_q(q) ↦ UA_q(q) ↦
            user_q(q) ↦ roles_q(q)
  add_v: ∀u, q ·
            q ∈ STATE ∧
            u ∈ ℙ(USER) ∧ u ∩ U_q(q) = ∅
      ⇒ (U_q(q) ∪ u) ↦ S_q(q) ↦ UA_q(q) ↦ user_q(q) ↦
            roles_q(q) ∈ STATE      theorem

  del_t: deleteUsers ∈ STATE × ℙ(USER) → STATE
  del_d: ∀u, q ·
            q ∈ STATE ∧
            u ∈ ℙ(USER) ∧ u ⊆ U_q(q)
      ⇒ deleteUsers(q ↦ u) = (U_q(q)\u) ↦ S_q(q) ↦ (u ⩤ UA_q(q)) ↦
            (user_q(q) ⩥ u) ↦ roles_q(q)
  del_v: ∀u, q ·
            q ∈ STATE ∧ u ∈ ℙ(USER) ∧ u ⊆ U_q(q)
      ⇒ (U_q(q)\u) ↦ S_q(q) ↦ (u ⩤ UA_q(q)) ↦ (user_q(q) ⩥ u) ↦
            roles_q(q) ∈ STATE      theorem

  rev: ∀u, q ·
            q ∈ STATE ∧
            u ∈ ℙ(USER) ∧ u ∩ U_q(q) = ∅
      ⇒ deleteUsers(addUsers(q ↦ u) ↦ u) = q      theorem      >  reversibility
  com1: ∀q, u1, u2 ·
            q ∈ STATE ∧
            u1 ∈ ℙ(USER) ∧ u1 ∩ U_q(q) = ∅ ∧
            u2 ∈ ℙ(USER) ∧ u2 ∩ U_q(q) = ∅ ∧
            u1 ∩ u2 = ∅
      ⇒ addUsers(addUsers(q ↦ u1) ↦ u2) =
            addUsers(addUsers(q ↦ u2) ↦ u1)      theorem      >  commutativity
  com2: ∀q, u1, u2 ·
            q ∈ STATE ∧
            u1 ∈ ℙ(USER) ∧ u1 ⊆ U_q(q) ∧
            u2 ∈ ℙ(USER) ∧ u2 ⊆ U_q(q) ∧
            u1 ∩ u2 = ∅
        ⇒ deleteUsers(deleteUsers(q ↦ u1) ↦ u2) =
            deleteUsers(deleteUsers(q ↦ u2) ↦ u1)      theorem
END
```

D.4 Context `rbac_sessionhandling`

```
CONTEXT rbac_sessionhandling
EXTENDS
  rbac_state
CONSTANTS
  createSessions
  destroySessions
  mapUserSessions
  unmapUserSessions
AXIOMS
  crt_t: createSessions ∈ STATE × ℙ(SESSION) → STATE
  crt_d: ∀q, s ·
            q ∈ STATE ∧
            s ∈ ℙ(SESSION) ∧ finite(s) ∧ s ∩ S_q(q) = ∅
        ⇒ createSessions(q ↦ s) = U_q(q) ↦ (S_q(q) ∪ s) ↦ UA_q(q) ↦
            user_q(q) ↦ (roles_q(q) ∪ (s × {∅}))
  crt_v: ∀q, s ·
            q ∈ STATE ∧
            s ∈ ℙ(SESSION) ∧ finite(s) ∧ s ∩ S_q(q) = ∅
        ⇒ U_q(q) ↦ (S_q(q) ∪ s) ↦ UA_q(q) ↦ user_q(q) ↦
            (roles_q(q) ∪ (s × {∅})) ∈ STATE      theorem
  dty_t: destroySessions ∈ STATE × ℙ(SESSION) → STATE
  dty_d: ∀q, s ·
            q ∈ STATE ∧
            s ⊆ S_q(q)
        ⇒ destroySessions(q ↦ s) = U_q(q) ↦ (S_q(q)\s) ↦ UA_q(q) ↦
            (s ⩤ user_q(q)) ↦ (s ⩤ roles_q(q))
  dty_v: ∀q, s ·
            q ∈ STATE ∧
            s ⊆ S_q(q)
        ⇒ U_q(q) ↦ (S_q(q)\s) ↦ UA_q(q) ↦ (s ⩤ user_q(q)) ↦
            (s ⩤ roles_q(q)) ∈ STATE      theorem

  rev1: ∀q, s ·
            q ∈ STATE ∧
```

$\quad\quad s \in \mathbb{P}(\mathtt{SESSION}) \land \mathtt{finite}(s) \land s \cap \mathtt{S\_q}(q) = \varnothing$
$\quad \Rightarrow \mathtt{destroySessions}(\mathtt{createSessions}(q \mapsto s) \mapsto s) = q$ theorem

mp_t: $\mathtt{mapUserSessions} \in \mathtt{STATE} \times (\mathtt{SESSION} \rightarrowtail \mathtt{USER}) \rightarrow \mathtt{STATE}$

mp_d: $\forall q, su \cdot$
$\quad\quad q \in \mathtt{STATE} \land$
$\quad\quad su \in \mathtt{S\_q}(q) \setminus \mathrm{dom}(\mathtt{user\_q}(q)) \rightarrowtail \mathtt{U\_q}(q)$
$\quad\quad (\forall s \cdot s \in \mathrm{dom}(su) \Rightarrow \mathtt{roles\_q}(q)(s) \subseteq \mathtt{UA\_q}(q)[\{su(s)\}])$
$\quad \Rightarrow \mathtt{mapUserSessions}(q \mapsto su) = \mathtt{U\_q}(q) \mapsto \mathtt{S\_q}(q) \mapsto \mathtt{UA\_q}(q) \mapsto$
$\quad\quad (\mathtt{user\_q}(q) \cup su) \mapsto \mathtt{roles\_q}(q)$

mp_v: $\forall q, su \cdot$
$\quad\quad q \in \mathtt{STATE} \land$
$\quad\quad su \in \mathtt{S\_q}(q) \setminus \mathrm{dom}(\mathtt{user\_q}(q)) \rightarrowtail \mathtt{U\_q}(q)$
$\quad\quad (\forall s \cdot s \in \mathrm{dom}(su) \Rightarrow \mathtt{roles\_q}(q)(s) \subseteq \mathtt{UA\_q}(q)[\{su(s)\}])$
$\quad \Rightarrow \mathtt{U\_q}(q) \mapsto \mathtt{S\_q}(q) \mapsto \mathtt{UA\_q}(q) \mapsto (\mathtt{user\_q}(q) \cup su) \mapsto$
$\quad\quad \mathtt{roles\_q}(q) \in \mathtt{STATE}$ theorem

ump_t: $\mathtt{unmapUserSessions} \in \mathtt{STATE} \times (\mathtt{SESSION} \rightarrowtail \mathtt{USER}) \rightarrow \mathtt{STATE}$

ump_d: $\forall q, su \cdot$
$\quad\quad q \in \mathtt{STATE} \land$
$\quad\quad su \subseteq \mathtt{user\_q}(q)$
$\quad \Rightarrow \mathtt{unmapUserSessions}(q \mapsto su) = \mathtt{U\_q}(q) \mapsto \mathtt{S\_q}(q) \mapsto \mathtt{UA\_q}(q) \mapsto$
$\quad\quad (\mathtt{user\_q}(q) \setminus su) \mapsto \mathtt{roles\_q}(q)$

ump_v: $\forall q, su \cdot$
$\quad\quad q \in \mathtt{STATE} \land$
$\quad\quad su \subseteq \mathtt{user\_q}(q)$
$\quad \Rightarrow \mathtt{U\_q}(q) \mapsto \mathtt{S\_q}(q) \mapsto \mathtt{UA\_q}(q) \mapsto (\mathtt{user\_q}(q) \setminus su) \mapsto \mathtt{roles\_q}(q)$
$\quad\quad \in \mathtt{STATE}$ theorem

rev2: $\forall q, su \cdot$
$\quad\quad q \in \mathtt{STATE} \land$
$\quad\quad su \in \mathtt{S\_q}(q) \setminus \mathrm{dom}(\mathtt{user\_q}(q)) \rightarrowtail \mathtt{U\_q}(q)$
$\quad\quad (\forall s \cdot s \in \mathrm{dom}(su) \Rightarrow \mathtt{roles\_q}(q)(s) \subseteq \mathtt{UA\_q}(q)[\{su(s)\}])$
$\quad \Rightarrow \mathtt{unmapUserSessions}(\mathtt{mapUserSessions}(q \mapsto su) \mapsto su) = q$
theorem

END

D.5 Context rbac_rolehandling

CONTEXT rbac_rolehandling

EXTENDS

rbac_state

CONSTANTS

assignRolesToUsers

revokeRolesFromUsers

```
  activateRoles
  deactivateRoles
AXIOMS
  aur_t: assignRolesToUsers ∈ STATE × ℙ(USER × ROLE) → STATE
  aur_d: ∀q, ua ·
            q ∈ STATE ∧
            ua ⊆ (U_q(q) × ROLE)\UA_q(q)
        ⇒ assignRolesToUsers(q ↦ ua) = U_q(q) ↦ S_q(q) ↦
            (UA_q(q) ∪ ua) ↦ user_q(q) ↦ roles_q(q)
  aur_v: ∀q, ua ·
            q ∈ STATE ∧
            ua ⊆ (U_q(q) × ROLE)\UA_q(q)
        ⇒ U_q(q) ↦ S_q(q) ↦ (UA_q(q) ∪ ua) ↦ user_q(q) ↦ roles_q(q)
            ∈ STATE     theorem
  rur_t: revokeRolesFromUsers ∈ STATE × ℙ(USER × ROLE) → STATE
  rur_d: ∀q, ua ·
            q ∈ STATE ∧
            ua ⊆ UA_q(q) ∧
            (∀u · u ∈ dom(ua) ⇒ ua[{u}] ∩
                (⋃r0 · r0 ∈ roles_q(q)[user_q(q)⁻¹[{u}]] | r0) = ∅
            )
        ⇒ revokeRolesFromUsers(q ↦ ua) = U_q(q) ↦ S_q(q) ↦
            (UA_q(q)\ua) ↦ user_q(q) ↦ roles_q(q)
  rur_v: ∀q, ua ·
            q ∈ STATE ∧
            ua ⊆ UA_q(q) ∧
            (∀u · u ∈ dom(ua) ⇒ ua[{u}] ∩
                (⋃r0 · r0 ∈ roles_q(q)[user_q(q)⁻¹[{u}]] | r0) = ∅
            )
        ⇒ U_q(q) ↦ S_q(q) ↦ (UA_q(q)\ua) ↦ user_q(q) ↦ roles_q(q)
            ∈ STATE     theorem

  rev1: ∀q, ua ·
            q ∈ STATE ∧
            ua ⊆ (U_q(q) × ROLE)\UA_q(q)
        ⇒ revokeRolesFromUsers(assignRolesToUsers(q ↦ ua) ↦ ua) = q     theorem

  acr_t: activateRoles ∈ STATE × ℙ(SESSION × ROLE) → STATE
  acr_d: ∀q, sr ·
            q ∈ STATE ∧
            sr ⊆ user_q(q); UA_q(q) ∧ finite(sr) ∧
            (∀s · s ∈ dom(user_q(q)) ⇒ roles_q(q)(s) ∩ sr[{s}] = ∅)
        ⇒ activateRoles(q ↦ sr) = U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦
            user_q(q) ↦
            {s ↦ r |
              s ∈ dom(roles_q(q)) ∧
              r = (roles_q(q)(s) ∪ sr[ {s} ∩ dom(user_q(q)) ])
          }
  acr_v: ∀q, sr ·
          q ∈ STATE ∧
          sr ⊆ user_q(q); UA_q(q) ∧ finite(sr) ∧
          (∀s · s ∈ dom(user_q(q)) ⇒ roles_q(q)(s) ∩ sr[{s}] = ∅)
      ⇒ U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦ user_q(q) ↦
          {s ↦ r |
              s ∈ dom(roles_q(q)) ∧
              r = (roles_q(q)(s) ∪ sr[ {s} ∩ dom(user_q(q)) ])
          } ∈ STATE      theorem
  dar_t: deactivateRoles ∈ STATE × ℙ(SESSION × ROLE) → STATE
  dar_d: ∀q, sr ·
          q ∈ STATE ∧
          sr ⊆ {s ↦ r | s ∈ dom(roles_q(q)) ∧ r ∈ roles_q(q)(s)}
      ⇒ deactivateRoles(q ↦ sr) = U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦
          user_q(q) ↦
          {s ↦ r |
              s ∈ dom(roles_q(q)) ∧
              r = (roles_q(q)(s)\sr[ {s} ])
          }
  dar_v: ∀q, sr ·
          q ∈ STATE ∧
          sr ⊆ {s ↦ r | s ∈ dom(roles_q(q)) ∧ r ∈ roles_q(q)(s)}
      ⇒ U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦ user_q(q) ↦
          {s ↦ r |
              s ∈ dom(roles_q(q)) ∧
              r = (roles_q(q)(s)\sr[ {s} ])
          } ∈ STATE      theorem

  rev2: ∀q, sr ·
          q ∈ STATE ∧
          sr ⊆ user_q(q); UA_q(q) ∧ finite(sr) ∧
          (∀s · s ∈ dom(user_q(q)) ⇒ roles_q(q)(s) ∩ sr[{s}] = ∅)
      ⇒ deactivateRoles(activateRoles(q ↦ sr) ↦ sr) = q      theorem
END
```

D.6 Context rbac_conditions

```
CONTEXT rbac_conditions
EXTENDS
  rbac_state
CONSTANTS
  cond_UR
  cond_UM
  cond_SR
  cond_SM
  sod
AXIOMS
  aUR_t: cond_UR ∈ STATE × USER × ROLE → BOOL
  aUR_d: ∀ q, u, r ·
           q ∈ STATE ∧
           u ∈ USER ∧
           r ∈ ROLE
      ⇒ cond_UR(q ↦ u ↦ r) = bool(
           (∃ r1 · r1 ∈ ROLE ∧
                (r1 ↦ r) ∈ RH ∧
                (u ↦ r1) ∈ UA_q(q)
           )
      )

  aUM_t: cond_UM ∈ STATE × USER × OBJECT × OPERATION → BOOL
  aUM_d: ∀ q, u, o, op ·
           q ∈ STATE ∧
           u ∈ USER ∧
           o ∈ OBJECT ∧ op ∈ OPERATION
      ⇒ cond_UM(q ↦ u ↦ o ↦ op) = bool(
           (∃ r · r ∈ ROLE ∧ (r ↦ o) ∈ dom(M) ∧
                op ∈ M(r ↦ o) ∧
                cond_UR(q ↦ u ↦ r) = TRUE
           )
      )

  aSR_t: cond_SR ∈ STATE × SESSION × ROLE → BOOL
  aSR_d: ∀ q, s, r ·
           q ∈ STATE ∧
           s ∈ SESSION ∧
           r ∈ ROLE
      ⇒ cond_SR(q ↦ s ↦ r) = bool(
           s ∈ dom(roles_q(q)) ∧
           (∃ r1 · r1 ∈ ROLE ∧
                (r1 ↦ r) ∈ RH ∧
                r1 ∈ roles_q(q)(s)
           )
      )

  aSM_t: cond_SM ∈ STATE × SESSION × OBJECT × OPERATION → BOOL
  aSM_d: ∀ q, s, o, op ·
           q ∈ STATE ∧
```

```
            s ∈ SESSION ∧
            o ∈ OBJECT ∧ op ∈ OPERATION
        ⇒ cond_SM(q ↦ s ↦ o ↦ op) = bool(
            (∃ r · r ∈ ROLE ∧ (r ↦ o) ∈ dom(M) ∧
                op ∈ M(r ↦ o) ∧
                cond_SR(q ↦ s ↦ r) = TRUE
            )
        )

  sod_t:  sod ∈ STATE × USER × ROLE → BOOL
  sod_d:  ∀ q, u, r ·
            q ∈ STATE ∧
            u ∈ USER ∧
            r ∈ ROLE
        ⇒ sod(q ↦ u ↦ r) = bool(
            (∀ r1 · r1 ∈ ROLE ∧
                (u ↦ r1) ∈ UA_q(q) ⇒ ¬((r ↦ r1) ∈ RE)
            )
        )
END
```

D.7 Context rbac

```
CONTEXT rbac
EXTENDS
  rbac_userhandling
  rbac_sessionhandling
  rbac_rolehandling
  rbac_conditions
END
```

D.8 Context healthcare_context

```
CONTEXT healthcare_context
EXTENDS
  rbac
CONSTANTS
  Employee      > roles
  Manager
  Doctor
  Nurse
  Receptionist
  Patient
  MedicalManager
  MedicalTeam
  ReferredDoctor
  UserAdmin

  view      > operations
  add
  modify
  access
  enter
  create
  update
  sign

  OldMedicalRecords      > objects
  RecentMedicalRecords
  PrivateNotes
  Prescriptions
  PatientPersonalInfo
  PatientFinancialInfo
  PatientMedicalInfo
  CarePlan
  Appointment
  ProgressNotes
  LegalAgreement
  Bills
  Uo
  UAo

  u1     > user of initial state

  RH_base      > required only for construction
  RE_base
AXIOMS
  axm1: partition(ROLE,
          {Employee}, {Manager}, {Doctor}, {Nurse}, {Receptionist},
          {Patient}, {MedicalManager}, {MedicalTeam},
          {ReferredDoctor}, {UserAdmin}
      )
```

```
axm2: partition(OBJECT,
        {OldMedicalRecords}, {RecentMedicalRecords},
        {PrivateNotes}, {Prescriptions}, {PatientPersonalInfo},
        {PatientFinancialInfo}, {PatientMedicalInfo}, {CarePlan},
        {Appointment}, {ProgressNotes}, {LegalAgreement}, {Bills},
        {Uo}, {UAo}
    )
axm3: partition(OPERATION,
        {view}, {add}, {modify}, {access}, {enter}, {create},
        {update}, {sign}
    )
axm4: partition(USER, {u1})

axm5: RH_base = {
        Doctor ↦ Nurse,
        Nurse ↦ Employee,
        Manager ↦ MedicalManager,
        MedicalManager ↦ Receptionist,
        Receptionist ↦ Employee
    } ∪ id
axm6: RH = RH_base; RH_base; RH_base
axm7: Manager ↦ Employee ∈ RH ∧
        Manager ↦ Manager ∈ RH      theorem

axm8: RE_base = {
        Patient ↦ Employee,
        Patient ↦ ReferredDoctor,
        Patient ↦ UserAdmin,
        Doctor ↦ Manager,
        Doctor ↦ Receptionist
    }
axm9: RE = RE_base ∪ RE_base⁻¹

axm10: M = {
        Doctor ↦ OldMedicalRecords ↦ {view},
        Patient ↦ OldMedicalRecords ↦ {view},
        Manager ↦ OldMedicalRecords ↦ {enter},
        Nurse ↦ OldMedicalRecords ↦ {access},
        Doctor ↦ RecentMedicalRecords ↦ {view, add},
        Nurse ↦ RecentMedicalRecords ↦ {view},
        Patient ↦ RecentMedicalRecords ↦ {view},
        Manager ↦ RecentMedicalRecords ↦ {enter},
        Doctor ↦ PrivateNotes ↦ {view, add},
        Doctor ↦ Prescriptions ↦ {view, modify},
        Patient ↦ Prescriptions ↦ {view},
          Manager ↦ PatientPersonalInfo ↦ {access},
          Manager ↦ PatientFinancialInfo ↦ {access},
          Manager ↦ PatientMedicalInfo ↦ {access},
          Manager ↦ CarePlan ↦ {update},
          Nurse ↦ CarePlan ↦ {view},
          Receptionist ↦ Appointment ↦ {create},
          Nurse ↦ ProgressNotes ↦ {add},
          Patient ↦ LegalAgreement ↦ {sign},
          Patient ↦ Bills ↦ {view},
          UserAdmin ↦ Uo ↦ {update},
          UserAdmin ↦ UAo ↦ {update}
      }
END
```

D.9 Context healthcare_generic

```
MACHINE healthcare_generic
SEES healthcare_context
VARIABLES
  state
INVARIANTS
  inv1: state ∈ CONSISTENT_STATE
  inv2: state ∈ STATE    theorem
EVENTS
INITIALISATION ≙
  begin
    act1: state := {u1} ↦ ∅ ↦ {u1 ↦ UserAdmin} ↦ ∅ ↦ ∅
  end
createUser ≙
  any
    s     > acting session
    u     > user to be created
  where
    typ1: s ∈ SESSION
    typ2: u ∈ USER\U_q(state)
    cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
  then
    act1: state := addUsers(state ↦ {u})
  end
destroyUser ≙
  any
    s     > acting session
    u     > user to be deleted
  where
    typ1: s ∈ SESSION
    typ2: u ∈ U_q(state)
    cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
  then
    act1: state := deleteUsers(
             destroySessions(
                 state ↦
                 dom(user_q(state) ▷ {u})
             ) ↦
             {u}
          )
  end
login ≘
  any
    s      > session to be created
    u      > assigned user
  where
    typ1: s ∈ SESSION\S_q(state)
    typ2: u ∈ U_q(state)
  then
    act1: state := mapUserSessions(
             createSessions(state ↦ {s}) ↦
             {s ↦ u}
          )
  end
logout ≘
  any
    s      > session that logs out
  where
    typ1: s ∈ S_q(state)
  then
    act1: state := destroySessions(
             unmapUserSessions(
                 state ↦
                 {s ↦ user_q(state)(s)}
             ) ↦
             {s}
          )
  end
activateRole ≘
  any
    s      > acting session
    r       > role to be activated
  where
    typ1:  $s \in$ S_q(state)
    typ2:  $r \in$ ROLE\roles_q(state)(s)
    cnd1:  $r \in$ (user_q(state); UA_q(state))[{s}]      > user may activate r
  then
    act1:  state := activateRoles( state $\mapsto$ {s $\mapsto$ r} )
  end
deactivateRole $\widehat{=}$
  any
    s       > acting session
    r       > role to be deactivated
  where
    typ1:  $s \in$ S_q(state)
    typ2:  $r \in$ roles_q(state)(s)
  then
    act1:  state := deactivateRoles( state $\mapsto$ {s $\mapsto$ r} )
  end
assignGenericRole $\widehat{=}$
  any
    u       > user
    r       > role to be assigned
  where
    typ1:  $u \in$ U_q(state)
    typ2:  $r \in$ ROLE\UA_q(state)[{u}]
    cnd1:  sod(state $\mapsto$ u $\mapsto$ r) = TRUE
  then
    act1:  state := assignRolesToUsers( state $\mapsto$ {u $\mapsto$ r} )
  end
revokeGenericRole $\widehat{=}$
  any
    u       > user
    r       > role to be revoked
  where
    typ1:  $u \in$ U_q(state)
    typ2:  $r \in$ UA_q(state)[{u}]
  then
```

```
    act1: state := revokeRolesFromUsers(
            deactivateRoles(
                state ↦
                { s0 · s0 ∈ user_q(state)⁻¹[{u}] ∧
                    r ∈ roles_q(state)(s0) | s0 ↦ r}
            ) ↦
            {u ↦ r}
        )
  end
END
```

D.10 Context healthcare

```
MACHINE healthcare
REFINES healthcare_generic
SEES healthcare_context
VARIABLES
  state     > all invariants of healthcare_generic hold
EVENTS
INITIALISATION ≙
  extends
    INITIALISATION
  begin
    act1: state := {u1} ↦ ∅ ↦ {u1 ↦ UserAdmin} ↦ ∅ ↦ ∅
  end
createUser ≙
  extends
    createUser
  any
    s
    u
  where
    typ1: s ∈ SESSION
    typ2: u ∈ USER\U_q(state)
    cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
  then
    act1: state := addUsers(state ↦ {u})
  end
destroyUser ≙
  extends
    destroyUser
  any
    s
    u
  where
    typ1: s ∈ SESSION
    typ2: u ∈ U_q(state)
    cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
  then
    act1: state := deleteUsers(
             destroySessions(
                 state ↦ dom(user_q(state) ▷ {u})
             ) ↦ {u}
        )
  end
assignRole ≙
  extends
    assignGenericRole
  any
    s     ▷  action session
    u     ▷  target user
    r     ▷  role to be assigned
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    typ3: r ∈ ROLE\UA_q(state)[{u}]
    cnd1: sod(state ↦ u ↦ r) = TRUE
    cnd2: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE
  then
    act1: state := assignRolesToUsers( state ↦ {u ↦ r} )
  end
revokeRole ≙
  extends
    revokeGenericRole
  any
    s     ▷  acting session
    u     ▷  target user
    r     ▷  role to be revoked
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    typ3: r ∈ UA_q(state)[{u}]
    cnd1: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE
  then
    act1: state := revokeRolesFromUsers(
              deactivateRoles(
                  state ↦
                  { s0 · s0 ∈ user_q(state)⁻¹[{u}] ∧
                      r ∈ roles_q(state)(s0) | s0 ↦ r}
              ) ↦
              {u ↦ r}
          )
  end
login ≙
  extends
    login
  any
    s
    u
  where
    typ1: s ∈ SESSION\S_q(state)
    typ2: u ∈ U_q(state)
  then
    act1: state := mapUserSessions(
              createSessions(state ↦ {s}) ↦
              {s ↦ u}
          )
  end
logout ≙
  extends
    logout
  any
    s
  where
    typ1: s ∈ S_q(state)
  then
    act1: state := destroySessions(
              unmapUserSessions(
                  state ↦
                  {s ↦ user_q(state)(s)}
              ) ↦ {s}
          )
  end
activateRole ≙
  extends
    activateRole
  any
    s
    r
  where
    typ1: s ∈ S_q(state)
    typ2: r ∈ ROLE\roles_q(state)(s)
    cnd1: r ∈ (user_q(state);UA_q(state))[{s}]
  then
    act1: state := activateRoles(state ↦ {s ↦ r})
  end
deactivateRole ≙
  extends
    deactivateRole
  any
    s
    r
  where
    typ1: s ∈ S_q(state)
    typ2: r ∈ roles_q(state)(s)
  then
    act1: state := deactivateRoles(state ↦ {s ↦ r})
  end

assignReferredDoctorRole ≙
  refines
    assignGenericRole
  any
    s      >  acting session
    u      >  target user
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: sod(state ↦ u ↦ ReferredDoctor) = TRUE
    cnd2: access_SR(state ↦ s ↦ Doctor) = TRUE
    cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE
    cnd4: ReferredDoctor ∉ UA_q(state)[{u}]
  with
    r: r = ReferredDoctor
  then
```

```
    act1: state := assignRolesToUsers(
              state ↦ {u ↦ ReferredDoctor}
          )
  end
revokeReferredDoctorRole ≙
  refines
    revokeGenericRole
  any
    s     >  acting session
    u     >  target user
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: access_SR(state ↦ s ↦ Doctor) = TRUE
    cnd2: ReferredDoctor ∈ UA_q(state)[{u}]
  with
    r: r = ReferredDoctor
  then
    act1: state := revokeRolesFromUsers(
              deactivateRoles(
                  state ↦
                  {s0 · s0 ∈ user_q(state)^{-1}[{u}] ∧
                      ReferredDoctor ∈ roles_q(state)(s0)
                      | s0 ↦ ReferredDoctor}
              ) ↦
              {u ↦ ReferredDoctor}
          )
  end

assignPatientRole ≙
  refines
    assignGenericRole
  any
    s     >  acting session
    u     >  target user
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: sod(state ↦ u ↦ Patient) = TRUE
    cnd2: access_SR(state ↦ s ↦ Receptionist) = TRUE
    cnd4: Patient ∉ UA_q(state)[{u}]
  with
```

```
    r: r = Patient
  then
    act1: state := assignRolesToUsers(
              state ↦ {u ↦ Patient}
          )
  end
revokePatientRole ≙
  refines
    revokeGenericRole
  any
    s      > acting session
    u      > target user
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: access_SR(state ↦ s ↦ Receptionist) = TRUE
    cnd2: Patient ∈ UA_q(state)[{u}]
  with
    r: r = Patient
  then
    act1: state := revokeRolesFromUsers(
              deactivateRoles(
                  state ↦
                  {s0 · s0 ∈ user_q(state)^-1[{u}] ∧
                      Patient ∈ roles_q(state)(s0)
                      | s0 ↦ Patient}
              ) ↦
              {u ↦ Patient}
          )
  end

assignMedicalTeamRole ≙
  refines
    assignGenericRole
  any
    s      > acting session
    u      > target user
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: sod(state ↦ u ↦ MedicalTeam) = TRUE
    cnd2: access_SR(state ↦ s ↦ MedicalManager) = TRUE
    cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE ∨
            access_UR(state ↦ u ↦ Nurse) = TRUE
    cnd4: MedicalTeam ∉ UA_q(state)[{u}]
  with
    r: r = MedicalTeam
  then
    act1: state := assignRolesToUsers(
            state ↦ {u ↦ MedicalTeam}
        )
  end
revokeMedicalTeamRole ≙
  refines
    revokeGenericRole
  any
    s     > acting session
    u     > target user
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: access_SR(state ↦ s ↦ MedicalManager) = TRUE
    cnd2: MedicalTeam ∈ UA_q(state)[{u}]
  with
    r: r = MedicalTeam
  then
    act1: state := revokeRolesFromUsers(
            deactivateRoles(
                state ↦
                {s0 · s0 ∈ user_q(state)^-1[{u}] ∧
                    MedicalTeam ∈ roles_q(state)(s0)
                    | s0 ↦ MedicalTeam}
            ) ↦
            {u ↦ MedicalTeam}
        )
  end
END
```

List of Abbreviations

ABAC	Attribute-based Access Control
ABAM	Attribute-based Access Matrix
ACI	Access Control Information
ACM	Access Control Matrix
ADF	Access Decision Control Facility
ADT	Abstract Data Type
AEF	Access Enforcement Facility
API	Application Programming Interface
ARBAC	Administrative Role-based Access Control
ASID	Address Space Identifier
AVC	Access Vector Cache
BLP	Bell/La-Padula
CFM	Confinement Flow Model
CPU	Central Processing Unit
DAC	Discretionary Access Control
DFA	Deterministic Finite Automaton
DLM	Decentralized Label Model
EID	Entity Identifier
FAF	Flexible Authorization Framework
GFAC	Generalized Framework for Access Control
GTRBAC	Generalized Temporal Role-Based Access Control
GUI	Graphical User Interface
HIS	Health Information System
HRU	Harrison Ruzzo Ullman
IBAC	Identity-Based Access Control
IFG	Information Flow Graph
IPC	Inter Process Communication
IPPC	Inter Policy Process Communication
ITC	Inter Thread Communication
LAN	Local Area Network
LOC	Lines of Code
MAC	Mandatory Access Control
MLS	Multi Level Security
MMU	Memory Management Unit
NIST	National Institute of Standards and Technology
OASIS	Advancing Open Standards for the Information Society
OS	Operating System
PAP	Policy Administration Point
PARBAC	Parameterized Role-based Access Control
PCB	Process Control Block
PDP	Policy Decision Point

PEP	Policy Enforcement Point
PM	Policy Machine
PPCB	Policy Process Control Block
PPID	Policy Process Identifier
RBAC	Role-based Access Control
RSBAC	Rule Set Based Access Control
RT	Role-based Trust-management
RTE	Runtime Environment
SACMAT	Symposium on Access Control Models and Technologies
SBAC	Status-based Access Control
SE	Security Enhanced
SPID	Security Policy Identifier
SPM	Schematic Protection Model
SRK	Storage Root Key
TAID	Transaction Identifier
TAM	Types Access Matrix
TCB	Trusted Computing Base
TE	Type Enforcement
TID	Thread Identifier
TM	Trust-management
TPM	Trusted Platform Module
TPS	Trusted Persistent Storage
TRBAC	Temporal Role-based Access Control
VPFS	Virtual Private File System
VPN	Virtual Private Network
WAN	Wide Area Network
XACML	eXtensible Access Control Markup Language
XML	eXtensible Markup Language

List of Figures

List of Tables

Bibliography

[1] The Lex & Yacc Page. `http://www.compilertools.net/`. Accessed June 2013.

[2] The MEMPHIS Tree Builder & Tree Walker Tool. `http://memphis.compilertools.net/`. Accessed June 2013.

[3] A. Ott and S. Ievlev and H. W. Klöpping. The RSBAC Introduction. `http://books.rsbac.org`. Accessed April 2012.

[4] M. D. Abrams, K. W. Eggers, L. J. L. Padula, and I. M. Olson. A Generalized Framework for Access Control: An Informal Description. In *Proceedings of the 13th National Computer Security Conference*. NIST/NCSC, Washington, DC, USA, 1990.

[5] J.-R. Abrial. A Formal Approach to Large Software Construction. In J. Snepscheut, editor, *Mathematics of Program Construction*, volume 375 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 1989. ISBN 978-3-540-51305-6.

[6] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010. ISBN 978-0-521-89556-9. 1–586 pp.

[7] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT) - Special Section on VSTTE 2008*, 12 (6):447–466, Nov. 2010. ISSN 1433-2779.

[8] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer Verlag, 1996. ISBN 3-540-61929-1.

[9] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing Process Isolation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, MSPC '06, pages 1–10. ACM, New York, NY, USA, 2006. ISBN 1-59593-578-9.

[10] M. A. Al-Kahtani and R. Sandhu. Induced Role Hierarchies with Attribute-based RBAC. In *Proceedings of the Eighth ACM symposium on Access Control Models and Technologies*, SACMAT '03, pages 142–148. ACM, New York, NY, USA, 2003. ISBN 1-58113-681-1.

[11] Altova. Altova® UModel® 2012 Enterprise Edition. `http://www.altova.com/download/umodel/uml_tool_enterprise.html`, 2012. Accessed June 2012.

[12] P. E. Ammann and R. S. Sandhu. Extending the Creation Operation in the Schematic Protection Model. In *Proceedings of the 6th Annual Computer Security Applications Conference*, pages 340–348. IEEE Computer Society Press, Los Alamitos, CA, USA, Dec. 1990. ISBN 978-0-8186-2105-5.

[13] P. E. Ammann and R. S. Sandhu. Safety Analysis for the Extended Schematic Protection Model. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 87–97. IEEE Computer Society Press, Washington, DC, USA, May 1991. ISBN 0-8186-2168-0.

[14] P. Amthor, A. Fischer, and W. E. Kühnhauser. Analyse von Zugriffssteuerungssystemen (in German). In P. Horster and P. Schartner, editors, *D·A·CH Security 2009*, pages 49–61. syssec Verlag, 2009. ISBN 978-3-00-027488-6.

[15] P. Amthor, W. E. Kühnhauser, and A. Pölck. Model-based Safety Analysis of SELinux Security Policies. In P. Samarati, S. Foresti, J. Hu, and G. Livraga, editors, *In Proceedings of 5th International Conference on Network and System Security*, NSS '11, pages 208–215. IEEE Computer Society Press, Los Alamitos, CA, USA, 2011. ISBN 978-1-4577-0458-1.

[16] P. Amthor, W. E. Kühnhauser, and A. Pölck. Heuristic Safety Analysis of Access Control Models. In *In Proceedings of the 18th ACM Symposium on Access control Models and Technologies*, SACMAT '13, pages 137–148. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-1950-8.

[17] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, Hanscom AFB, Bedford, MA, USA, 1972. URL `http://seclab.cs.ucdavis.edu/projects/history/papers/ande72a.pdf`. Also available as Vol. I, DITCAD-758206. Vol. II DITCAD-772806.

[18] B. Parducci, H. Lockhart, and R. Levison. OASIS eXtensible Access Control Markup Language (XACML) TC Version 2.0. `https://www.oasis-open.org/`, Feb. 2005. Accessed July 2012.

[19] N. Baracaldo, A. Masoumzadeh, and J. Joshi. A Secure, Constraint-Aware Role-Based Access Control Interoperation Framework. In *Proceedings of the 5th International Conference on Network and System Security*, NSS '11, pages 200–207. IEEE Computer Society Press, Los Alamitos, CA, USA, 2011. ISBN 978-1-4503-1950-8.

[20] S. Barker. The Next 700 Access Control Models or a Unifying Meta-Model? In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 187–196. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-537-6.

[21] S. Barker. Personalizing Access Control by Generalizing Access Control. In *Proceedings of the 15th ACM Symposium on Access Control Models and Technologies*, SACMAT '10, pages 149–158. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0049-0.

[22] S. Barker, G. Boella, D. M. Gabbay, and V. Genovese. A Meta-model of Access Control in a Fibred Security Language. *Studia Logica, Special Issue: New Ideas in Applied Logic*, 92(3):437–477, 2009. ISSN 0039-3215.

[23] S. Barker and P. J. Stuckey. Flexible Access Control Policy Specification with Constraint Logic Programming. *ACM Transactions on Information and System Security*, 6(4):501–546, Nov. 2003. ISSN 1094-9224.

[24] M. Becker, C. Fournet, and A. Gordon. Design and Semantics of a Decentralized Authorization Language. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 3–15. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2819-8.

[25] M. Y. Becker and P. Sewell. Cassandra: Flexible Trust Management, Applied to Electronic Health Records. *IEEE Computer Security Foundations Workshop*, 0:139, 2004. ISBN 0-7695-2169-X. ISSN 1063-6900.

[26] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations (Vol.I). Technical Report AD-770 768, MITRE, Bedford, Massachusetts, USA, Nov. 1973. URL `http://www.dtic.mil/dtic/tr/fulltext/u2/770768.pdf`.

[27] D. E. Bell and L. J. LaPadula. Secure Computer Systems: A Refinement of the Mathematical Model. Technical Report AD-780 528, MITRE, Bedford, Massachusetts, USA, Apr. 1974. URL `http://www.dtic.mil/dtic/tr/fulltext/u2/780528.pdf`.

[28] D. E. Bell and L. J. LaPadula. Secure Computer System: Unified Exposition and Multics Interpretation. Technical Report AD-A023 588, MITRE, Bedford, Massachusetts, USA, Mar. 1976. URL `http://www.dtic.mil/dtic/tr/fulltext/u2/a023588.pdf`.

[29] A. Belokosztolszki and K. Moody. Meta-Policies for Distributed Role-Based Access Control Systems. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks)*, POLICY '02, pages 106–115. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1611-4.

[30] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An Access Control Model Supporting Periodicity Constraints and Temporal Reasoning. *ACM Transactions on Database Systems*, 23(3):231–285, Sept. 1998. ISSN 0362-5915.

[31] E. Bertino, B. Catania, M. L. Damiani, and P. Perlasca. GEO-RBAC: A spatially aware RBAC. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies*, SACMAT '05, pages 29–37. ACM, New York, NY, USA, 2005. ISBN 1-59593-045-0.

[32] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A Logical Framework for Reasoning About Access Control Models. In *Proceedings of the 6th ACM Symposium on Access Control Models and Technologies*, SACMAT '01, pages 41–52. ACM, New York, NY, USA, 2001. ISBN 1-58113-350-2.

[33] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A System to Specify and Manage Multipolicy Access Control Models. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, POLICY '02, pages 116–27. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1611-4.

[34] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A Logical Framework for Reasoning about Access Control Models. *ACM Transactions on Information and System Security*, 6(1):71–127, Feb. 2003. ISSN 1094-9224.

[35] K. Biba. Integrity Considerations for Secure Computer Systems. Technical Report ESD-TR-76-372, MITRE, Bedford, Massachusetts, USA, Apr. 1977. URL `www.dtic.mil/dtic/tr/fulltext/u2/a039324.pdf`.

[36] C. Bidan and V. Issarny. Dealing with Multi-policy Security in Large Open Distributed Systems. In *Proceedings of the 5th European Symposium on Research in Computer Security*, pages 51–66. Springer-Verlag, London, UK, 1998. ISBN 3-540-65004-0.

[37] P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. A Modular Approach to Composing Access Control Policies. In *Proceedings of the 7th ACM Conference on Computer and Communications Security*, CCS '00, pages 164–173. ACM, New York, NY, USA, 2000. ISBN 1-58113-203-4.

[38] P. Bonatti and P. Samarati. Regulating Service Access and Information Release on the Web. In *Proceedings of the 7th ACM conference on Computer and communications security*, CCS '00, pages 134–143. ACM, New York, NY, USA, 2000. ISBN 1-58113-203-4. URL `http://doi.acm.org/10.1145/352600.352620`.

[39] P. A. Bonatti and P. Samarati. A Uniform Framework for Regulating Service Access and Information Release on the Web. *Journal of Computer Security*, 10(3):241–271, Sept. 2002. ISSN 0926-227X.

[40] T. Boswell. Specification and Validation of a Security Policy Model. In J. C. Woodcock and P. G. Larsen, editors, *FME '93: Industrial-Strength Formal Methods*, volume 670 of *Lecture Notes in Computer Science*, pages 42–51. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-56662-5.

[41] D. F. Brewer and M. J. Nash. The Chinese Wall Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 206–214. IEEE Computer Society, Los Alamitos, CA, USA, May 1989.

[42] N. Broberg and D. Sands. Flow Locks: Towards a Core Calculus for Dynamic Flow Policies. In P. Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 180–196. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-33095-0.

[43] N. Broberg and D. Sands. Flow-sensitive Semantics for Dynamic Information Flow Policies. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 101–112. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-645-8.

[44] N. Broberg and D. Sands. Paralocks: Role-based Information Flow Control and Beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 431–444. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-479-9.

[45] A. D. Brucker and B. Wolff. A Verification Approach to Applied System Security. *International Journal on Software Tools for Technology Transfer (STTT) - Special Section on Formal Methods for Industrial Critical Systems*, 7(3):233–247, June 2005. ISSN 1433-2779.

[46] C. Bryce, W. E. Kühnhauser, R. Amouroux, and M. Lopéz. CWASAR: A European Infrastructure for Secure Electronic Commerce. *Journal of Computer Security - Special issue on security in the World Wide Web*, 5(3):225–235, 1997.

[47] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

[48] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM)*, SPSM '11. ACM, New York, NY, USA, Oct. 2011. ISBN 978-1-4503-1000-0.

[49] S. Bugiel, S. Heuser, and A.-R. Sadeghi. Towards a Framework for Android Security Modules: Extending SE Android Type Enforcement to Android Middleware. Technical Report TUD-CS-2012-0231, System Security Lab / CASED, 2012. URL `http://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/flaskdroid_tr.pdf`.

[50] C. Stüble. The PERSEUS Security Framework. `http://www.perseus-os.org`. Accessed June 2012.

[51] P. Cao, E. W. Felten, and K. Li. Implementation and Performance of Application-controlled File Caching. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94. USENIX Association, Berkeley, CA, USA, 1994.

[52] J. Carter. Using GConf as an Example of How to Create an Userspace Object Manager. *2007 Security Enhanced Linux Symposium-SELinux*, 2007.

[53] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. System-wide Compaction and Specialization of the Linux Kernel. *SIGPLAN Notices*, 40:95–104, June 2005. ISSN 0362-1340.

[54] Citrix. XenClient XT. `http://www.citrix.com/English/ps2/products/subfeature.asp?contentID=2315434`. Accessed May 2012.

[55] CLEARSY, Aix-en-Provence, France. *Atelier B - Proof Obligations Reference Manual*, 2011. URL `http://www.tools.clearsy.com/index.php5?title=Documents`.

[56] R. Cocker. Porting NSA Security Enhanced Linux to Hand-held Devices. In *In Proceedings of the Linux Symposium*. Ottawa, Ontario, Canada, July 2003.

[57] I. D. Craig. *Formal Models of Operating System Kernels*. Springer London, 2007. ISBN 978-1-8462-8375-8. 1-333 pp.

[58] I. D. Craig. *Formal Refinement for Operating System Kernels*. Springer London, 2007. ISBN 978-1-84628-966-8. 1-332 pp.

[59] I. F. Cruz, R. Gjomemo, B. Lin, and M. Orsini. A Constraint and Attribute Based Security Framework for Dynamic Role Assignment in Collaborative Environments. In E. Bertino and J. B. D. Joshi, editors, *CollaborateCom*, volume 10 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 322–339. Springer, 2008. ISBN 978-3-642-03353-7.

[60] F. Cuppens and C. Saurel. Specifying a Security Policy: A Case Study. In *Proceedings of the Computer Security Foundations Workshop*. IEEE Computer Society, Kenmare, Ireland, 1996. ISBN 0-8186-7522-5. ISSN 1063-6900.

[61] D. F. Ferraiolo, S. Gavrila, and W. Jansen. Interagency Report - Policy Machine: Features, Architecture, and Specification. `http://csrc.nist.gov/pm/references-library.html`. Accessed June 2013.

[62] D. S. D. Krafzig, K. Banke. *Enterprise SOA: Service Oriented Architecture Best Practices*. Upper Saddle River, NJ, Prentice Hall, 2005. ISBN 0-13-146575-9.

[63] E. Damiani, S. di Vimercati, and P. Samarati. New Paradigms for Access Control in Open Environments. *International Symposium on Signal Processing and Information Technology*, 0:540–545, 2005. ISBN 0-7803-9313-9.

[64] M. L. Damiani, E. Bertino, B. Catania, and P. Perlasca. GEO-RBAC: A Spatially Aware RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 10 (1), Feb. 2007. ISSN 1094-9224.

[65] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In M. Sloman, E. Lupu, and J. Lobo, editors, *Policies for Distributed Systems and Networks*, volume 1995 of *Lecture Notes in Computer Science*, pages 18–38. Springer Berlin / Heidelberg, 2001. ISBN 3-540-41610-2.

[66] K. De Bosschere. Memory Footprint Reduction for Embedded Systems. In *Proceedings of the 11th International Workshop on Software & Compilers for Embedded Systems*, SCOPES '08, page 31. ACM, New York, NY, USA, 2008.

[67] D. E. Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–242, May 1976. ISSN 0001-0782.

[68] D. E. Denning and T. F. Lunt. A Multilevel Relational Data Model. *IEEE Symposium on Security and Privacy*, 0:220–234, 1987. ISSN 1540-7993.

[69] P. J. Denning. Third Generation Computer Systems. *ACM Computing Surveys*, 3(4): 175–216, Dec. 1971. ISSN 0360-0300.

[70] J. Dobson and J. McDermid. A Framework for Expressing Models of Security Policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 229–239, may 1989. ISBN 0-8186-1939-2.

[71] DP. Event-B – Industrial Projects. `http://wiki.event-b.org/index.php/Industrial_Projects`. Accessed August 2012.

[72] DP. Rodin User's Handbook, Version 2.4. `http://handbook.event-b.org/release-2012-04-04/html/`. Accessed May 2012.

[73] DSD/GCSB (Australia), CSE (Canada), DCSSI (France), BSI (Germany), ITPA (Japan), NNCSA (The Netherlands), MAPCCN (Spain), CESG (UK), NSA/NIST (USA). *Common Criteria for Information Technology Security Evaluation, Version 3.1, Revision 3*, July 2009.

[74] P. Efstathopoulos and E. Kohler. Manageable Fine-Grained Information Flow. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 301–313. ACM, New York, NY, USA, Apr. 2008. ISBN 978-1-60558-013-5.

[75] H.-D. Ehrich, M. Gogolla, and U. W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen - Eine Einführung in die Theorie*. Leitfäden und Monographien der Informatik. B. G. Teubner Stuttgart, 1989. ISBN 978-3-519-02266-4. I-IX, 1-263 pp.

[76] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985. ISBN 0387137181. 1-306 pp.

[77] M. Evered and S. Bögeholz. A Case Study in Access Control Requirements for a Health Information System. In *Proceedings of the 2nd Workshop on Australasian Information Security, Conferences in Research and Practice in Information Technology, Vol. 32*, ACSW Frontiers '04, pages 53–61. Australian Computer Society, Inc., Darlinghurst, Australia, 2004.

[78] B. Fabian, S. Gürses, M. Heisel, T. Santen, and H. Schmidt. A Comparison of Security Requirements Engineering Methods. *Requirements Engineering*, 15(1):7–40, Mar. 2010. ISSN 0947-3602.

[79] G. Faden. Solaris Trusted Extensions – Architectural Overview. `http://www.3c2controller.net/project/truetrue/solaris10/security/tx/TrustedExtensionsArch.pdf`, Apr. 2006. Sun/Oracle White Paper, Accessed June 2013.

[80] M. Felderer, B. Agreiter, and R. Breu. Evolution of Security Requirements Tests for Service-centric Systems. In *Proceedings of the Third International Conference on Engineering Secure Software and Systems*, ESSoS'11, pages 181–194. Springer-Verlag, Berlin, Heidelberg, 2011. ISBN 978-3-642-19124-4.

[81] D. F. Ferraiolo and V. Atluri. A Meta Model for Access Control: Why is it needed and Is it even possible to achieve? In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*, SACMAT '08, pages 153–154. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-129-3.

[82] D. F. Ferraiolo, V. Atluri, and S. Gavrila. The Policy Machine: A Novel Architecture and Framework for Access Control Policy Specification and Enforcement. *Journal of Systems Architecture*, 57(4):412–424, Apr. 2011. ISSN 1383-7621.

[83] D. F. Ferraiolo, J. F. Barkley, and D. R. Kuhn. A Role-Based Access Control Model and Reference Implementation within a Corporate Intranet. *ACM Transactions on Information and System Security (TISSEC) - Special Issue on Role-Based Access Control*, 2(1):34–64, Feb. 1999. ISSN 1094-9224.

[84] D. F. Ferraiolo, S. Gavrila, V. Hu, and D. R. Kuhn. Composing and Combining Policies under the Policy Machine. In *Proceedings of the tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, pages 11–20. ACM, New York, NY, USA, 2005. ISBN 1-59593-045-0.

[85] D. F. Ferraiolo, S. Gavrila, and W. Jansen. Enabling an Enterprise-Wide, Data-Centric Operating Environment. *Computer*, 46(4):94–96, 2013. ISSN 0018-9162.

[86] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control.* Information Security and Privacy Series. Artech House, 2007. 381 pp. Second Edition, ISBN 978-1-59693-113-8.

[87] D. F. Ferraiolo and R. Kuhn. Role-Based Access Controls. In *15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[88] D. F. Ferraiolo, R. Kuhn, and R. Sandhu. RBAC Standard Rationale: Comments on "A Critique of the ANSI Standard on Role-Based Access Control". *IEEE Security and Privacy*, 5(6):51–53, Nov. 2007. ISSN 1540-7993.

[89] D. F. Ferraiolo, J. M. Voas, and G. F. Hurlburt. A Matter of Policy. *IT Professional*, 14(2):4–7, 2012.

[90] N. Feske and C. Helmuth. A Nitpicker's guide to a minimal-complexity secure GUI. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 85–94. IEEE Press, Washington, DC, USA, 2005. ISBN 0-7695-2461-3.

[91] A. Fischer and W. E. Kühnhauser. Efficient Algorithmic Safety Analysis of HRU Security Models. In S. Katsikas and P. Samarati, editors, *Proc. International Conference on Security and Cryptography (SECRYPT 2010)*, pages 49–58. SciTePress, 2010.

[92] S. N. Foley. A Model for Secure Information Flow. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 248–258. IEEE Computer Society, May 1989. ISBN 0-8186-1939-2.

[93] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151. ACM, New York, NY, USA, 1996. ISBN 1-880446-82-0.

[94] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. Larus, and S. Levi. Language Support for fast and Reliable Message Based Communication in Singularity OS. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190. ACM, New York, NY, USA, 2006. ISBN 1-59593-322-0.

[95] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995. ISBN 978-0-201-63361-0.

[96] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A Virtual Machine-based Platform for Trusted Computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 193–206. ACM, New York, NY, USA, 2003. ISBN 1-58113-757-5.

[97] M. Gasser. *Building a Secure Computer System.* van Nostrand Reinhold, 1988. ISBN 0-442-23022-2.

[98] L. Giuri and P. Iglio. Role templates for content-based access control. In *RBAC '97: Proceedings of the Second ACM Workshop on Role-based Access Control*, pages 153–159. ACM Press, New York, NY, USA, 1997. ISBN 0-89791-985-8.

[99] M. I. Gofman, C. Ramakrishnan, S. D. Stoller, and P. Yang. Parameterized RBAC and ARBAC Policies for a Small Health Care Facility. `http://www.cs.stonybrook.edu/~stoller/parbac/healthcare.txt`, 2009. Accessed August 2011.

[100] J. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society, Apr. 1982.

[101] G. S. Graham and P. J. Denning. Protection: Principles and Practice. In *AFIPS '72 (Spring): Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, pages 417–429. ACM, New York, NY, USA, 1972.

[102] J. Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[103] R. Guerraoui and M. Kapalka. The Theory of Transactional Memory. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (97):83–105, Feb. 2009.

[104] J. D. Guttman, A. L. Herzog, and J. D. Ramsdell. Information Flow in Operating Systems: Eager Formal Methods. In *Workshop on Issues in the Theory of Security (WITS)*, 2003.

[105] U. Halfmann and W. E. Kühnhauser. Embedding Security Policies Into a Distributed Computing Environment. *Operating Systems Review*, 33(2):51–64, Apr. 1999. ISSN 0163-5980.

[106] M. A. Harrison and W. L. Ruzzo. Monotonic Protection Systems. In R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, editors, *Foundations of Secure Computation*, pages 337–365. Academic Press, 1978.

[107] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. On Protection in Operating Systems. *Operating Systems Review, Special Issue for the 5th Symposium on Operating Systems Principles*, 9(5):14–24, Nov. 1975.

[108] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, Aug. 1976. ISSN 0001-0782.

[109] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews. Code Compaction of an Operating System Kernel. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 283–298. IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2764-7.

[110] C. Helmuth, A. Warg, and N. Feske. Mikro-SINA - Hands-on Experiences with the Nizza Security Architecture. In P. Horster, editor, *D·A·CH Security 2005*. syssec Verlag, 2005. ISBN 3-00-015548-1.

[111] M. Herlihy. Transactional Memory: A Primer for Theorists. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, (98):123–138, June 2009.

[112] M. Hohmuth and H. Härtig. Pragmatic Nonblocking Synchronization for Real-Time Systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 217–230. USENIX Association, Berkeley, CA, USA, 2001. ISBN 1-880446-09-X.

[113] H. H. Hosmer. Metapolicies I. *SIGSAC Review*, 10:18–43, June 1992. ISSN 0277-920X.

[114] H. H. Hosmer. Metapolicies II. In *Proceedings of the 15th National Computer Security Conference*, pages 369–378. NIST/NCSC, United States Government Printing Office, 1992.

[115] V. C. Hu, D. A. Frincke, and D. F. Ferraiolo. The Policy Machine for Security Policy Management. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 494–506. Springer Verlag, London, UK, 2001. ISBN 3-540-42233-1.

[116] G. Hunt, M. Aiken, M. Fähndrich, C. Hawblitzel, S. Levi, B. Steendgaard, D. Tarditi, and T. Wobber. Sealing OS Processes to Improve Dependability and Safety. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 341–354. ACM, New York, NY, USA, Mar. 2007. ISBN 978-1-59593-636-3.

[117] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steendgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005. URL `http://research.microsoft.com/pubs/52716/tr-2005-135.pdf`.

[118] D. Hutter, M. Klusch, and M. Volkamer. Information Flow Analysis Based Security Checking of Health Service Composition Plans. In *Proceedings of the 1st European Conference on eHealth (ECEH)*, 2006.

[119] D. Hutter and M. Volkamer. Information Flow Control to Secure Dynamic Web Service Composition. In J. Clark, R. Paige, F. Polack, and P. Brooke, editors, *Security in Pervasive Computing*, volume 3934 of *Lecture Notes in Computer Science*, pages 196–210. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-33376-0.

[120] D. Hutter, M. Volkamer, M. Klusch, and A. Gerber. Provably Secure Execution of Composed Semantic Web Services. In *Proceedings of the 1st International Workshop on Privacy and Security in Agent-based Collaborative Environments*. AAMAS, 2006.

[121] T. Härder and A. Reuter. Principles of Transaction-oriented Database Recovery. *ACM Computing Survey*, 15(4):287–317, Dec. 1983. ISSN 0360-0300.

[122] H. Härtig. Security Architectures Revisited. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, EW 10, pages 16–23. ACM, New York, NY, USA, 2002.

[123] H. Härtig, M. Hohmuth, N. Feske, A. Lackorzynski, F. Mehnert, and M. Peter. The Nizza Secure-System Architecture. In *Proceedings 1st International Conference an Collaborative Computing*. IEEE Computer Society, Los Alamitos, CA, USA, Dec. 2005. ISBN 1-4244-0030-9.

[124] F. Innerhofer-Oberperfler and R. Breu. Using an Enterprise Architecture for IT Risk Management. In J. H. P. Eloff, L. Labuschagne, M. M. Eloff, and H. S. Venter, editors, *Proceedings of the Information Security South Africa Conference*, ISSA'06, pages 1–12. ISSA, Pretoria, South Africa, 2006. ISBN 1-86854-636-5.

[125] C. E. Irvine. The Reference Monitor Concept as a Unifying Principle in Computer Security Education. In *Proceedings of the IFIP TC11 WG 11.8 First World Conference on Information Security Education*, pages 27–37, 1999.

[126] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems*, 26(2): 214–260, June 2001. ISSN 0362-5915.

[127] S. Jajodia, P. Samarati, V. S. Subrahmanian, and E. Bertino. A Unified Framework for Enforcing Multiple Access Control Policies. *SIGMOD Record*, 26:474–485, June 1997. ISSN 0163-5808.

[128] M. Jastram. Rodin User's Handbook. `http://handbook.event-b.org/`. Version 2.4 2012, Accessed July 2012.

[129] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards Formal Verification of Role-Based Access Control Policies. *IEEE Transactions on Dependable Secure Computing*, 5:242–255, October 2008. ISSN 1545-5971.

[130] X. Jin, R. Krishnan, and R. Sandhu. A Unified Attribute-based Access Control Model Covering DAC, MAC and RBAC. In *Proceedings of the 26th Annual IFIP WG 11.3 Conference on Data and Applications Security and Privacy*, DBSec'12, pages 41–55. Springer-Verlag, Berlin, Heidelberg, 2012. ISBN 978-3-642-31539-8.

[131] J. B. D. Joshi, E. Bertino, U. Latif, and A. Ghafoor. A Generalized Temporal Role-Based Access Control Model. *IEEE Transactions on Knowledge and Data Engineering*, 17(1):4–23, Jan. 2005. ISSN 1041-4347.

[132] D. Kafura and D. Gracanin. An Information Flow Control Meta-Model. In *Proceedings of the 18th ACM Symposium on Access control Models and Technologies*, SACMAT '13, pages 101–112. ACM, New York, NY, USA, 2013. ISBN 978-1-4503-1950-8.

[133] KaiGai Kohei. Security Enhanced PostgreSQL. `http://code.google.com/p/sepgsql`. Accessed June 2013.

[134] S. Kamin. Routine run-time code generation. In *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 208–220. ACM, New York, NY, USA, 2003. ISBN 1-58113-751-6.

[135] B. Katt, T. Trojer, R. Breu, T. Schabetsberger, and F. Wozak. Meeting EHR Security Requirements: Authentication as a Security Service. In A. Brömme, T. Eymann, D. Hühnlein, H. Roßnagel, and P. Schmücker, editors, *perspeGKtive 2010 – Workshop Innovative und sichere Informationstechnologie für das Gesundheitswesen von morgen*, volume 174 of *LNI*, pages 103–110. GI, 2010.

[136] B. Kauer and M. Völp. *L4.Sec – Preliminary Microkernel Reference Manual.* Technische Universitat Dresden, 01062 Dresden, Germany, Oct. 2005. Version: 0.2, October 19, 2005.

[137] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an Operating-System Kernel. *Communications of the ACM*, 53 (6):107–115, June 2010. ISSN 0001-0782.

[138] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220. ACM, New York, NY, USA, 2009.

[139] D. R. Kuhn. Mutual Exclusion of Roles as a Means of Implementing Separation of Duty in Role-Based Access Control Systems. In *Proceedings of the Second ACM Workshop on Role-Based Access Control*, RBAC '97, pages 23–30. ACM, New York, NY, USA, 1997. ISBN 0-89791-985-8.

[140] W. E. Kühnhauser. On Paradigms for Security Policies in Multipolicy Environments. In *Proceedings of the 11th International Information Security Conference (IFIP/SEC '95)*, pages 421–435. Chapman & Hall, May 1995.

[141] W. E. Kühnhauser. Policy Groups. *Computers & Security*, 18(4):351–363, 1999.

[142] W. E. Kühnhauser and M. von Kopp Ostrowski. A Framework to Support Multiple Security Policies. In *Proceedings of the 7th Canadian Computer Security Symposium*, pages 301–322. Communications Security Establishment Press, Ottawa, Canada, May 1995.

[143] A. Kurmus, A. Sorniotti, and R. Kapitza. Attack Surface Reduction for Commodity OS Kernels: Trimmed Garden Plants May Attract Less Bugs. In *Proceedings of the 4th European Workshop on System Security*, EUROSEC '11, pages 6:1–6:6. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0613-3.

[144] W. Kühnhauser and A. Pölck. Towards Access Control Model Engineering. In S. Jajodia and C. Mazumdar, editors, *Information Systems Security*, volume 7093 of *Lecture Notes in Computer Science*, pages 379–382. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-25559-5.

[145] B. W. Lampson. Protection. In *5th Annual Princeton Conference on Information Sciences and Systems*, pages 437–443, Mar. 1971. Reprinted January, 1974: Protection. In *Operating Systems Review*, 8(1), pages 18–24.

[146] B. Lang, I. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A Multipolicy Authorization Framework for Grid Security. In *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 269–272. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2640-3.

[147] B. Lang, I. T. Foster, F. Siebenlist, R. Ananthakrishnan, and T. Freeman. A Flexible Attribute Based Access Control Method for Grid Computing. *Journal of Grid Computing*, 7(2):169–180, 2009. ISSN 1570-7873.

[148] L. J. LaPadula. A Rule-Set Approach to Formal Modeling of a Trusted Computer System. *Computer Systems*, 7(1):113–167, Jan. 1994. ISSN 0895-6340.

[149] J. Larus and C. Kozyrakis. Transactional Memory. *Communications of the ACM*, 51 (7):80–88, July 2008. ISSN 0001-0782.

[150] C.-H. Lee, M. C. Chen, and R.-C. Chang. HiPEC: High Performance External Virtual Memory Caching. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI '94. USENIX Association, Berkeley, CA, USA, 1994.

[151] J. R. Levine. *flex & bison - Text Processing Tools*. O'Reilly, 2009. ISBN 978-0-596-15597-1. 1-271 pp.

[152] N. Li, J.-W. Byun, and E. Bertino. A Critique of the ANSI Standard on Role-Based Access Control. *IEEE Security and Privacy*, 5(6):41–49, Nov. 2007. ISSN 1540-7993.

[153] N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 58–73. Springer Verlag, London, UK, 2003. ISBN 3-540-00389-4.

[154] N. Li and J. C. Mitchell. RT: a Role-based Trust-management Framework. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX-III 2003)*, pages 201–212. IEEE Computer Society, 2003. ISBN 0-7695-1897-4.

[155] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust-Management Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 114–130. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-7695-1543-6.

[156] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond Proof-of-compliance: Security Analysis in Trust Management. *Journal of the ACM*, 52(3):474–514, May 2005. ISSN 0004-5411.

[157] N. Li and M. V. Tripunitara. Security Analysis in Role-Based Access Control. *ACM Transactions on Information and System Security (TISSEC)*, 9(4):391–420, Nov. 2006. ISSN 1094-9224.

[158] N. Li, M. V. Tripunitara, and Z. Bizri. On Mutually Exclusive Roles and Separation-of-Duty. *ACM Transactions on Information and System Security (TISSEC*, 10, May 2007. ISSN 1094-9224.

[159] N. Li and W. Winsborough. Beyond proof-of-compliance: Safety and Availability Analysis in Trust Management. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P 2003)*, pages 123–139. IEEE Computer Society, 2003. ISBN 0-7695-1940-7. ISSN 1081-6011.

[160] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed Credential Chain Discovery in Trust Management: Extended Abstract. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, CCS '01, pages 156–165. ACM, New York, NY, USA, 2001. ISBN 1-58113-385-5.

[161] J. Liedtke. On μ-Kernel Construction. *SIGOPS Operating Systems Review*, 29(5): 237–250, Dec. 1995. ISSN 0163-5980.

[162] J. Liedtke. On μ-Kernel Construction. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, 1995. ISBN 0-89791-715-4.

[163] J. Liedtke. Toward Real Microkernels. *Communications of the ACM*, 39(9):70–77, 1996. ISSN 0001-0782.

[164] LMP. Linux Man Pages. `http://linux.die.net/man/`. Accessed July 2012.

[165] P. A. Loscocco and S. D. Smalley. Integrating Flexible Support for Security Policies into the Linux Operating System. In C. Cole, editor, *2001 USENIX Annual Technical Conference*, pages 29–42, 2001. ISBN 1-880446-10-3.

[166] P. A. Loscocco and S. D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux. In *Proceedings of the 2001 Linux Symposium*, 2001.

[167] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckmann, and W. R. Shockley. The SeaView Security Model. *IEEE Transactions on Software Engineering*, 16(6):593–607, June 1990. ISSN 0098-5589.

[168] L. Martino, Q. Ni, D. Lin, and E. Bertino. Multi-domain and Privacy-aware Role Based Access Control in eHealth. In *Second International Conference on Pervasive Computing Technologies for Healthcare*, pages 131–134, 2008. ISBN 978-963-9799-15-8.

[169] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-0-7695-4035-1.

[170] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-013-5.

[171] J. M. McCune, N. Qu, Y. Li, A. Datta, V. D. Gligor, and A. Perrig. Efficient TCB Reduction and Attestation. Technical Report CMU-CyLab-09-003, CyLab Carnegie Mellon University, March 2009. URL `http://people.csail.mit.edu/costan/readings/oakland_papers/CMUCylab09003.pdf`.

[172] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. *SIGOPS Operating Systems Review*, 31(5):129–142, Oct. 1997. ISSN 0163-5980.

[173] A. C. Myers and B. Liskov. Complete, Safe Information Flow with Decentralized Labels. In *Proceedings of the 1998 IEEE Computer Society Symposium on Research in Security and Privacy (RSP)*, pages 186–197. IEEE Computer Society Press, Los Alamitos, CA, USA, May 1998.

[174] A. C. Myers and B. Liskov. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, Oct. 2000. ISSN 1049-331X.

[175] Y. Nakamura and Y. Sameshima. SELinux for Consumer Electronics Devices. In *In Proceedings of the Linux Symposium*, pages 125–134, 2008.

[176] P. Naldurg and R. K. R. SEAL: A Logic Programming Framework for Specifying and Verifying Access Control Models. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT 2011, pages 83–92. ACM, 2011. ISBN 978-1-4503-0688-1.

[177] National Security Agency – Central Security Service. Security-Enhanced Linux. `http://www.nsa.gov/research/selinux/index.shtml`, 2010. Accessed June 2010.

[178] F. Neumann. Formale Spezifikation politikbezogener Funktionen einer Trusted Computing Base (in German). Master's thesis, Ilmenau University of Technology, Aug. 2012.

[179] G. Neumann and M. Strembeck. A Scenario-Driven Role Engineering Process for Functional RBAC Roles. In *Proceedings of the 7th ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 33–42. ACM, New York, NY, USA, 2002. ISBN 1-58113-496-7.

[180] Q. Ni and E. Bertino. xfACL: An Extensible Functional Language for Access Control. In *Proceedings of the 16th ACM Symposium on Access Control Models and Technologies*, SACMAT '11, pages 61–72. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0688-1.

[181] Q. Ni, E. Bertino, and J. Lobo. Risk-based Access Control Systems Built on Fuzzy Inferences. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 250–260. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-936-7.

[182] M. Nyanchama and S. Osborn. Modeling Mandatory Access Control in Role-based Security Systems. In *Proceedings of the 9th Annual IFIP TC11 WG11.3 Working Conference on Database Security IX : Status and Prospects*, pages 129–144. Chapman & Hall, Ltd., London, UK, 1996. ISBN 0-412-72920-2.

[183] Object Management Group (OMG). Unified Modeling Language (UML), v2.4.1, Infrastructure and Superstructure. `http://www.omg.org/spec/UML/2.4.1/`, 2011. Accessed June 2012.

[184] S. Oh, R. Sandhu, and X. Zhang. An Effective Role Administration Model Using Organization Structure. *ACM Transactions on Information and System Security (TISSEC)*, 9(2):113–137, May 2006. ISSN 1094-9224.

[185] Oracle. Oracle Label Security. http://www.oracle.com/technetwork/database/options/label-security. Accessed May 2012.

[186] Oracle. Oracle Solaris 10. http://www.oracle.com/us/products/servers-storage/solaris/solaris10/. Accessed May 2012.

[187] Oracle. Oracle Database Security Guide. http://docs.oracle.com/, 2011. 11g Release 1 (11.1), B28531-15, Accessed May 2012.

[188] S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security*, 3(2):85–106, May 2000. ISSN 1094-9224.

[189] A. Ott. The Role Compatibility Security Model. In *Nordic Workshop on Secure IT Systems 2002*, NordSec '02, Nov. 2002.

[190] A. Ott. Die Architektur des Linux-Sicherheitssystems Rule Set Based Access Control (RSBAC) – Sicherheits-Architektur (in German). *Linux Magazin*, (1):48–53, 2003. ISSN 1094-9224.

[191] A. Ott and S. Fischer-Hübner. The Rule Set Based Access Control (RSBAC) Framework for Linux. http://thehackademy.net/madchat/sysadm/linux/rsbac-framework.pdf, 2001.

[192] M. P. Papazoglou and W. J. Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal*, 16(3):389–415, 2007. ISSN 1066-8888.

[193] J. Park and R. Sandhu. The UCONABC Usage Control Model. *ACM Transactionson Information and System Security (TISSEC)*, 7(1):128–174, Feb. 2004. ISSN 1094-9224.

[194] H. Pennington. GConf: Manageable User Preferences. In *In Proceedings of the 2002 Ottawa Linux Symposium*. Ottawa, Canada, June 2002.

[195] B. Pfitzmann, J. Riordan, C. Stüble, M. Waidner, and A. Weber. The PERSEUS System Architecture. Technical Report RZ 3335 (93381) 04/09/01, Universität des Saarlandes, 2001. URL http://www.zurich.ibm.com/security/publications/2001/PRSWW01_PERSEUS_IBMRZ.pdf.

[196] M. Poletto, W. C. Hsieh, D. R. Engler, and M. F. Kaashoek. 'c and tcc: A language and compiler for dynamic code generation. *ACM Transactionson on Programming Languages and Systems (TOPLAS)*, 21(2):324–369, Mar. 1999. ISSN 0164-0925.

[197] Robert N. M. Watson. The TrustedBSD Project. http://www.trustedbsd.org/sebsd.html. Accessed April 2012.

[198] RPD. Event-B – Rodin Plug-ins. http://wiki.event-b.org/index.php/Rodin_Plug-ins. Accessed August 2012.

[199] M. J. Rutherford and A. L. Wolf. A Case for Test-code Generation in Model-driven Systems. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 377–396. Springer-Verlag New York, Inc., New York, NY, USA, 2003. ISBN 3-540-20102-5.

[200] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Sept. 2006. ISSN 0733-8716.

[201] A.-R. Sadeghi and C. Stüble. Taming "Trusted Platforms" by Operating System Design. In K.-J. Chae and M. Yung, editors, *Information Security Applications*, volume 2908 of *Lecture Notes in Computer Science*, pages 286–302. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-20827-3.

[202] R. Sailer, T. Jaeger, E. Valdez, R. Cáceres, R. Perez, S. Berger, J. Linwood, and G. L. Doorn. Building a MAC-based Security Architecture for the Xen Opensource Hypervisor. ACSAC '05, pages 276–285. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2461-3.

[203] R. Sailer, E. Valdez, T. Jaeger, R. Perez, L. van Doorn, J. L. Griffin, and S. Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC23511 (W0502-006), Feb. 2005. URL `http://domino.watson.ibm.com/library/CyberDig.nsf/papers/265C8E3A6F95CA8D85256FA1005CBF0F/$File/rc23511.pdf`.

[204] R. Sandhu. The Schematic Protection Model: Its Definition and Analysis for Acyclic Attenuating Schemes. *Journal of the ACM*, 35(2):404–432, 1988. ISSN 0004-5411.

[205] R. Sandhu, D. F. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards a Unified Standard. In *RBAC '00: Proceedings of the fifth ACM Workshop on Role-Based Access Control*, pages 47–63. ACM, New York, NY, USA, 2000. ISBN 1-58113-259-X.

[206] R. Sandhu and Q. Munawer. How to do Discretionary Access Control Using Roles. In *Proceedings of the 3rd ACM Workshop on Role-Based Access Control*, RBAC '98, pages 47–54. ACM, New York, NY, USA, 1998. ISBN 1-58113-113-5.

[207] R. S. Sandhu. A Lattice Interpretation of the Chinese Wall Policy. In *Proceedings of the 15th National Computer Security Conference*, pages 329–339. NIST/NCSC, United States Government Printing Office, 1992.

[208] R. S. Sandhu. Lattice-Based Enforcement of Chinese Walls. *Computers & Security*, 11 (9):753–763, Dec. 1992. ISSN 0167-4048.

[209] R. S. Sandhu. The Typed Access Matrix Model. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, SP '92, pages 122–136. IEEE Computer Society, Washington, DC, USA, May 1992. ISBN 0-8186-2825-1.

[210] R. S. Sandhu. Role-Based Access Control. In *Advances in Computers*. Academic Press, 1994.

[211] R. S. Sandhu. Role Hierarchies and Constraints for Lattice-Based Access Controls. In *Proceedings of the 4th European Symposium on Research in Computer Security*, ESORICS '96, pages 65–79. Springer-Verlag, London, UK, 1996. ISBN 3-540-61770-1.

[212] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, Feb. 1996. ISSN 0018-9162.

[213] A. Schaad, J. Moffett, and J. Jacob. The Role-Based Access Control System of a European Bank: A Case Study and Discussion. In *Proceedings of the sixth ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 3–9. ACM, New York, NY, USA, 2001. ISBN 1-58113-350-2.

[214] F. W. Schröer. The GENTLE Compiler Construction System – Handbook. `http://gentle.compilertools.net/book/`. Accessed June 2013.

[215] F. W. Schröer. *The GENTLE Compiler Construction System*. R. Oldenbourg, Munich and Vienna, 1997. ISBN 3-486-247034-4. 142 pp.

[216] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-591-5.

[217] A. Shabtai, Y. Fledel, and Y. Elovici. Securing Android-Powered Mobile Devices Using SELinux. *Security & Privacy*, 8(3):36–44, 2010. ISSN 1540-7993.

[218] A. B. Shaffer, M. Auguston, C. E. Irvine, and T. E. Levin. A Security Domain Model to Assess Software for Exploitable Covert Channels. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, PLAS '08, pages 45–56. ACM, New York, NY, USA, 2008. ISBN 978-1-59593-936-4.

[219] H.-b. Shen and F. Hong. An Attribute-Based Access Control Model for Web Services. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 74–79. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2736-1.

[220] L. Singaravelu, C. Pu, H. Härtig, and C. Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 161–174. ACM, New York, NY, USA, 2006. ISBN 1-59593-322-0.

[221] L. A. Slevin and A. Macfie. Role Based Access Control for a Medical Database. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications*, SEA '07, pages 226–233. ACTA Press, Anaheim, CA, USA, 2007. ISBN 978-0-88986-706-2.

[222] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *19th Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.

[223] S. D. Smalley. Configuring the SELinux Policy. Technical Report 02-007, NAI Labs, Jan. 2003. URL `http://classes.soe.ucsc.edu/cmps122/Spring07/selinux-paper.pdf`.

[224] S. D. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a Linux Security Module. Technical Report 01-043, NAI Labs, May 2002. URL `http://www.nsa.gov/research/_files/publications/implementing_selinux.pdf`.

[225] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium*, SSYM'99, pages 123–139. USENIX Association, Berkeley, CA, USA, 1999.

[226] W. Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2009. ISBN 978-0-13-603337-0.

[227] U. Steinberg and B. Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222. ACM, New York, NY, USA, 2010. ISBN 978-1-60558-577-2.

[228] S. D. Stoller, P. Yang, M. Gofman, and C. R. Ramakrishnan. Symbolic Reachability Analysis for Parameterized Administrative Role Based Access Control. In *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*, SACMAT '09, pages 165–174. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-537-6.

[229] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient Policy Analysis for Administrative Role Based Access Control. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 445–455. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-703-2.

[230] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. RBAC and ARBAC Policies for a Small Health Care Facility. `http://www.cs.sunysb.edu/~stoller/ccs2007/healthcare.txt`, 2007. Accessed August 2011.

[231] M. Strembeck. Scenario-Driven Role Engineering. *IEEE Security and Privacy*, 8(1): 28–35, Jan. 2010. ISSN 1540-7993.

[232] Sybase. System Administration Guide: Volume2. `http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc31652.1570/pdf/java.pdf`. Adaptive Server Enterprise 15.7, DC31644-01-1570-01, Accessed May 2011.

[233] M. Tripunitara and N. Li. The Foundational Work of Harrison-Ruzzo-Ullman Revisited. *IEEE Transactions on Dependable and Secure Computing*, 10:28–39, 2013. ISSN 1545-5971.

[234] M. V. Tripunitara and N. Li. A Theory for Comparing the Expressive Power of Access Control Models. *Journal of Computer Security*, 15(2):231–272, Apr. 2007. ISSN 0926-227X.

[235] Trusted Computing Group (TCG). TPM Main Specification Level 2 Version 1.2, Revision 116, Design Principles, Structures of the TPM, Commands. `http://www.trustedcomputinggroup.org`, 2011. Accessed June 2012.

[236] TUD:OS-Gruppe (TU Dresden Operating Systems). Fiasco. `http://os.inf.tu-dresden.de`. Accessed June 2013.

[237] TUD:OS-Gruppe (TU Dresden Operating Systems). L4 Runtime Environment. `http://os.inf.tu-dresden.de/L4Re/`. Accessed June 2012.

[238] U. Steinberg. NOVA Microhypervisor. `http://hypervisor.org/`. Accessed June 2013.

[239] Uni-Düsseldorf. The ProB Animator and Model Checker. `http://www.stups.uni-duesseldorf.de/ProB/`. Accessed August 2012.

[240] C. Vance and R. Watson. Security Enhanced BSD. Technical Report RR-06-04, Network Associates Laboratories Rockville, July 2003. URL `http://www.trustedbsd.org/sebsd-july2003.pdf`.

[241] S. D. C. d. Vimercati, P. Samarati, and S. Jajodia. Policies, Models, and Languages for Access Control. In *4th International Workshop on Databases in Networkes Information Systems (DNIS 2005)*, volume 3433/2005 of *LNCS*, pages 225–237. Springer, 2005. ISBN 978-3-540-25361-7.

[242] D. Volpano, C. Irvine, and G. Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996. ISSN 0926-227X.

[243] L. A. Wahsheh and J. Alves-Foss. Specifying and Enforcing a Multi-Policy Paradigm for High Assurance Multi-Enclave Systems. *Journal of High Speed Networks*, 15(3): 315–327, 2006. ISSN 0926-6801.

[244] E. F. Walsh. Application of the Flask Architecture to the X Window System Server. In *Proceedings of the 2007 SELinux Symposium*, 2007.

[245] L. Wang, D. Wijesekera, and S. Jajodia. A Logic-based Framework for Attribute based Access Control. In *Proceedings of the 2004 ACM Workshop on Formal Methods in Security Engineering*, FMSE '04, pages 45–55. ACM, New York, NY, USA, 2004. ISBN 1-58113-971-3.

[246] R. Watson, B. Feldman, A. Migus, and C. Vance. Design and Implementation of the TrustedBSD MAC Framework. *DARPA Information Survivability Conference and Exposition*, 1:38–49, 2003. ISBN 0-7695-1897-4.

[247] R. Watson and C. Vance. The TrustedBSD MAC Framework: Extensible Kernel Access Control for FreeBSD 5.0. In *In USENIX Annual Technical Conference*, pages 285–296, 2003.

[248] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-508-8.

[249] C. Weinhold and H. Härtig. VPFS: Building a Virtual Private File System with a Small Trusted Computing Base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 81–93. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-013-5.

[250] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36, Oct. 2009. ISSN 0360-0300.

[251] E. Yuan and J. Tong. Attributed Based Access Control (ABAC) for Web Services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 561–569. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2409-5.

[252] G. Zanin and L. V. Mancini. Towards a Formal Model for Security Policies Specification and Validation in the SELinux System. In *Proceedings of the 9th ACM Symposium on Access Control Models and Technologies*, SACMAT '04, pages 136–145. ACM, New York, NY, USA, 2004. ISBN 1-58113-872-5.

[253] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making Information Flow Explicit in HiStar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278. USENIX Association, Berkeley, CA, USA, 2006. ISBN 1-931971-47-1.

[254] X. Zhang, Y. Li, and D. Nalla. An Attribute-based Access Matrix Model. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, SAC '05, pages 359–363. ACM, New York, NY, USA, 2005. ISBN 1-58113-964-0.