Privacy-Aware Dynamic Coalitions
A Formal Framework

vorgelegt von
Diplom-Informatiker
Nadim Sarrouh

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangen des akademischen Grades

Doktor der Ingenieurwissenschaften
– Dr.-Ing.–

genehmigte Dissertation

Promotionsausschuss:

Vorsitzende: Prof. Dr. Sabine Glesner
Gutachter: Prof. Dr. Uwe Nestmann
Gutachter: Prof Dr. Bernd Mahr
Gutacherin: Prof. Dr. Ina Schieferdecker

Tag der wissenschaftlichen Aussprache: 01. April 2014

Berlin 2014

D 83

Zusammenfassung

Diese Dissertation behandelt die Definition datenschutzkritischer, dynamischer Koalitionen anhand eines formalen Rahmenwerks, basierend auf dem *Abstract State Machine (ASM)* Formalismus. Zu diesem Zweck wurden zuerst elementare dynamische Koalitionen definiert und dann mit verschiedenen Zugriffskontrollmechanismen erweitert um Datenschutz-erzwingende Operationen in die Informationsaustauschprozesse der Koalitionen zu integrieren. Das resultierende Rahmenwerk besteht aus einer Anzahl von *ASM*-Modellen, welche sowohl elementare dynamische Koalitionen als auch dynamische Koalitionen mit *identitätsbasierter (IBAC), rollenbasierter (RBAC), attributbasierter (ABAC)* oder *vertrauensbasierter (TBAC)* Zugriffskontrolle definieren, wobei jede jeweils für dynamische Koalitionen mit unterschiedlichen Mitgliedschaftsdynamiken Anwendung finden. Die These dieser Arbeit konstatiert, dass das vorgeschlagene Rahmenwerk das Verständnis und die Entwicklung von Software für diese Koalitionen während der typischen Phasen eines Softwareentwicklung-Phasenmodells unterstützt: *Anforderungsanalyse, detaillierter Entwurf, Validierung* und *Dokumentation*. Die These wurde in zwei Fallstudien validiert, welche die Korrektheit der These durch die Anwendung des Rahmenwerks in zwei Szenarien aus dem Medizinsektor untersuchen. Die Ergebnisse zeigen, das die These für die beiden Fallstudien gültig ist und liefern außerdem Auskunft darüber, wie das Rahmenwerk benutzt werden kann um eine Brücke zu schlagen zwischen dem Verständnis von Softwareentwicklern und Domainexperten, wie etwa jene Ärzte die bei der Erstellung der hier präsentierten Fallstudienmodelle mitgewirkt haben.

Abstract

This dissertation deals with the definition of privacy-aware dynamic coalitions by means of a formal framework, based on the *Abstract State Machine (ASM)* formalism. To this end, basic dynamic coalitions were defined and then extended with various access control mechanisms in order to integrate privacy-enforcing operations into the coalition's information sharing processes. The resulting framework consists of a number of *ASM* models, which define basic dynamic coalitions as well as dynamic coalitions with *identity-based access control* (*IBAC*), *role-based access control* (*RBAC*), *attribute-based access control* (*ABAC*) and *trust-based access control (TBAC)* with each one applying to dynamic coalitions of different membership dynamics. The thesis of this work states, that the proposed framework supports the understanding and the development of software for these coalitions throughout the typical software engineering life cycle: *requirement capture*, *detailed design*, *validation* and *documentation*. The thesis is validated in two case studies, which investigate the correctness of the thesis through the application of the framework in two dynamic coalition scenarios taken from the medical sector. The results show that the thesis holds for the presented case studies and gives insight also into how the model may be used to bridge the gap of understanding between software engineers and domain experts such as the medical doctors who contributed in the creation of the case study models presented in this work.

Acknowledgements

This thesis is a result of my four years of research as part of the *SOAMED Research Training Group*. As member of the *Models and Theory of Distributed Systems Group* at *Technical Institute Berlin* I received my main supervision from its director Uwe Nestmann. First and foremost want to express my gratitude for this supervision through countless fruitful meetings and the perfect mixture of intellectual guidance and freedom, which motivated me to gain ground in a foreign field and at the same time develop responsibility to go my own ways. After four years I am confident to say that this supervision was suited best to my character and abilities. Furthermore, I want to thank the members of both research groups for the various meetings, presentations and conferences which were always productive and cooperative.

I want to thank my colleague and good friend Sebastian Bab for always being available for discussions and collaboration. Many of the contents of this thesis are directly or indirectly a result of these valuable conversations and joint research approaches. His companionship enriched my research from the first day of bouncing ideas to the last day in which he contributed to proof-reading the final drafts of my thesis.

My highest appreciation goes to Bernd Mahr for part-tacking in the supervision of my thesis. With his many years of experience in the field of logics and set theory he provided some most valuable input for the *Abstrat State Machine (ASM)*-based work at hand.

Last but not least I want to thank all my fellow colleagues from the *SOAMED research training group* who at all times gave me a sense of a comman journey and whom I wish all the best.

Contents

Chapter 1

Introduction & Motivation

1.1 Problem Statement

Dynamic coalitions are a phenomenon of modern information technologies and the new possibilities of interaction and communication emerging from its architectures, such as *cloud-computing*, *service-oriented architectures (SOA)* or *online social networks (OSN)*. These technologies allow companies, organizations, individuals or other autonomous agents to build coalitions for sharing their information or resources in order to achieve a common goal. These coalitions are dynamic in the sense, that they are temporary and may change during their existence. Dynamic coalitions form as a response to either an acute need or a long term goal, eventually however, when the goal is reached, they will dissolve. Coalition membership is also dynamic, allowing agents to join or leave the coalition during runtime. A common example of dynamic coalitions is the collaboration of emergency services, military and civilian institutions with the goal to contain acute crises, such as terrorist attacks or epidemics. Temporary cooperations between companies in order to exploit a current market chance may also be regarded as dynamic coalitions.

Other than the structure of these coalitions their actual processes are of critical interest. They pose both the motivation for the collaboration in the first place as well as the greatest challenges for the achievement of the common goal: The dynamics of coalition membership raise valid questions concerning the practicability and feasibility of processes in dynamic coalitions. Assessing and enforcing other non-quantitative quality measures , such as privacy, access control or trust may become non-trivial, difficult tasks regarding the underlying architectures which are distributed and heterogeneous by nature.

The degree of dynamics may vary from coalition to coalition meaning that changes in the coalitions structure happen more frequently or more often. Some coalitions are planned others just emerge. In some the members to join the coalition are to a great extent known before runtime in others the membership development is not predictable at all. The problem of how to support dynamic coalitions with IT-infrastructure is therefore highly dependent on the actual use case. Traditional software engineering projects used disregard the notion of a dynamic coalition and traditionally construct rather static software solutions which solve the requirements of certain scenarios only. However, many application domains, such as medicine or crisis management would prefer a rather open and dynamic solution approach in order to be able to respond to the dynamics of real-life. Certain software paradigms, such as *service-oriented architectures* or *cloud* promise to fulfill those needs. However, although they are to a certain extent implementation independent, they are programming paradigms, dealing with the software technical infrastructure to be

implemented.

This thesis on the other hand deals with the formalization of the notion of a dynamic coalition as a concept from a platform and implementation independent perspective. It is inspired by dynamic coalition scenarios from the medical field which may be supported by software infrastructures and in which agents may not only be computational agents but also individuals , such as doctors, nurses, patients, etc. In particular a certain subclass of dynamic coalitions is of interest, which this work refers to as *privacy-sensitive dynamic coalitions*. This type of coalition is characterized by the need for privacy-enforcing mechanisms which protect access on the shared data according to an agents specification. The framework presented here aims at providing an understanding of what a (privacy-sensitive) dynamic coalition is, how it is created and under which mechanisms it evolves and operates, resulting in a unique understanding of the matter at hand. To that end a formal definition of the notion of a dynamic coalition will be provided, which finally may also facilitate the modeling and development of software and applications for such scenarios.

1.1.1 Privacy in Dynamic Coalitions

In dynamic coalitions non-functional properties like security and privacy are of high relevance. Consider the following example taken from the medical sector:

> After the rehabilitation process in a stationary clinic a stroke patient is discharged and the ongoing ambulant rehabilitation measures are now overlooked and managed by the patient's family doctor. This doctor receives information of the rehabilitation process in the clinic through an explicit doctor's letter containing all the treatment up to this date as well as recommendations for further ambulant treatment. The doctor then decides, which ambulant therapists the patient should see, e.g. a physio-therapist, ergo-therapist, logo-therapist and so on. He sets up therapy orders and sends the patient to the therapists in question. After some therapy sessions the patient has another stroke and is delivered to the emergency room or stroke unit without any ability to speak or communicate. The emergency doctor now wants to access the information generated in the former ambulant rehabilitation process in order to investigate what caused the stroke or just to understand what actually happened thereby optimizing their emergency treatment.

Supporting scenarios like this with software technical means does not in itself pose huge efforts by the software engineer. However, the information that is shared and processed is *privacy-sensitive*, which means that the disclosure of the information has to be constrained according to moral, ethic and juristic requirements. These privacy of information constraints are often handled through *access control policies* of various kinds, which users or administrators define. However in dynamic coalitions scenarios it is often not previously known, which agents will collaborate at what time and if there are any administrators at all. Thus, on the one hand it is often difficult to foresee, which rules have to be considered for the policy beforehand and on the other hand a static and pre-defined set of policy rules may prevent a coalition with highly dynamic changes of membership and structure from working properly. Consider the above-mentioned scenario: If the policy designer does not foresee that after a second stroke another stroke-unit might need to access previously collected information on the patient, the policy will not support this access and therefore hinder the correct execution of the stroke-unit's treatment process.

The traditional way to solve this problem is to undergo process analysis in which all possible processes which might take place are recorded on the basis of which a

software architecture and policies may be provided to support these processes. However, in dynamic, distributed and complex scenarios which todays network technologies allow for, it often proves impossible to capture all possible processes rendering this static approach not suitable. For example, although in the past many patient treatment processes have been recorded and standardized in medicine (so called clinical pathways) since then practice has shown that the complexity of a patient's needs are very often not met by those standards and the treatment process has to deviate frequently from and sometimes even completely leave the standardized process pattern. Thus, medical experts wish strongly for a dynamic process and policy definition in the medical domain. To this end, dynamic access control concepts have to be integrated with dynamic coalitions, allowing for the dynamic extension and adaption of access control policies to the needs of the moment.

Ensuring that this dynamic approach to privacy is still secure and correct, according to the legal and institutional requirements is not a trivial task. At this point an integrated formal understanding of dynamic coalitions, as well as their access control mechanisms and their processes, would help to validate and verify the adherence to the privacy requirements, thereby equipping software engineers with tools to address those requirements on an early stage of the development.

1.1.2 Formal Modeling of Dynamic Coalitions

Because of these challenges an understanding of the structural properties, as well as the dynamic properties of these coalitions, is not only essential for effectively supporting them by developing fitting architectures: it is also critical in order to grasp the above-mentioned non-quantitative quality measure of privacy. Thus, in recent years various groups have started efforts to formalize basic concepts of dynamic coalitions. Specialized workshops, such as *FAVO (Formal Aspects of Virtual Organizations)*[1] see the development of verifiable formal models of dynamic coalitions as their main motivation. A position paper of these formalization efforts is to be found in [BF08].

The goal of this thesis is to provide a formal modeling framework in order to capture the basic concepts of a dynamic coalition in general as well as privacy-sensitive dynamic coalitions in particular. As illustrated by Bryans et al. in [BFJ+06] the relevant components of a dynamic coalition model may vary according to the perspective of the modeler. Thus, certain dimensions of dynamic coalitions have been identified, such as *coalition membership*, *information transfer* or *authorization structure*, each modeling the basic aspects relevant to the particular perspective. Joining this approach this thesis seeks a modeling perspective which not only formally captures structural properties but also the processes of dynamic coalitions thereby providing the means to validate and verify processes in dynamic coalitions as well as logical considerations concerning privacy and access control questions.

This thesis will show that the term "dynamic coalition" introduced above may be defined through the *abstract state machine (*ASM*)* [BS03] formalism which allows for the definition of static mathematical structures as well as for dynamic process attributes through means of a transition logic over these structures. The formal mathematical character of *ASMs* assures that a unique understanding of a dynamic coalition is provided and allows for the validation and verification of certain logical properties through means of simulation, testing or proof techniques including model checking respectively.

As mentioned above, in many domains, such as in the medical sector, the formation and creation of dynamic coalitions is closely connected to the compliance to privacy guidelines taken from law or institutional terms and conditions. In the

[1] See http://homepages.cs.ncl.ac.uk/jeremy.bryans/FAVO2011 for more information on *FAVO*

medical sector, which is the main subject of the case studies of this work, modern IT-concepts and mechanisms which allow for dynamic and adaptable processes, such as *SOA* or *Cloud*, are only used to a very limited extent because of strict and strong privacy guidelines. Therefore it is essential for software projects in the medical sector to start out with and integrate formal and verifiable models which also allow for the detailed definition of privacy requirements into each phase of '*the typical software engineering life cycle, namely*:

- **requirement capture** by constructing satisfactory *ground models*,
- **detailed design** by *stepwise refinement* of models to executable code,
- **validation** of models by their simulation, [...]
- **verification** of model properties by proof techniques or model checking,
- **documentation** for inspection, reuse and maintenance, by providing through the intermediate models explicit descriptions of the software structure and of the major design decisions." [BS03]

Thus, this work aims to provide a modeling framework for privacy-sensitive dynamic coalitions, integrating access control concepts, such as *identity-based access control* (*IBAC*), *role-based access control* (*RBAC*), *attribute-based access control* (ABAC) and *trust-based access control* (*TBAC*) with the dynamic coalition paradigm, each applied to coalitions of certain degrees of membership dynamics. The transition logic that *ASM* provides allows to view this created structure as a modifiable state, therefore enabling the formalization of the processes as transition rules in-between states. Such a framework will allow for the validation of privacy-policies in various scenarios as well as simulation of the behavior of collaborating agents in these scenarios, thereby identifying errors or incompleteness in policy or process definitions. By adding privacy aspects into process modeling the framework provides a modeling tool which exceed traditional process model techniques such as *BPMN*. The framework is furthermore intended to bridge the gap between software developers and domain experts, such as doctors from the medical field by providing a tool which may allow both parties to reason on common grounds: Since *ASM* is easily conceivable for software developers and yet may simplify software technical processes to their most important aspects, it might be used as a common language through which a unique understanding of the specification may be guaranteed. It will furthermore set the ground for formal verification of logical and privacy properties.

1.1.3 Dynamic Coalitions and SOA

This work was created in the course of the *SOAMED Research Training Group*, which investigates the scientific improvement of *service-oriented* concepts and technologies for application in medical domains.

> "In this situation, this Graduate School starts out with the idea to underpin the currently pragmatically focussed service-oriented approach with theoretical foundations by integrating established as well as emerging software engineering procedures. This approach aims at a decisive improvement of concepts, methods, and tool support for service-oriented system construction."

Therefore, the investigation of the *SOA* paradigm was the starting point of the research which led to this thesis. With the popularization of *SOA* and related

paradigms such as *Grid* or *Cloud* the notion of dynamic coalitions emerged, since it was now technologically possible to support the ad-hoc interactions needed for this kind of coalition. The software development paradigm *service oriented-architecture (*SOA*)* aims to achieve a dynamic binding between interacting software agents which may be accessed via a network, provide certain functionalities and may be combined or composed in order to create structured services [VVE10]. As in *component-based architecture* service agents collectively provide the complete functionality of a larger software application. This idea to loosely couple software components in order to provide the complete functionality of a larger software application mirrors the main motivation of dynamic coalition formation: the collaboration of interactive agents (computational or human) in order to achieve a common greater goal. It is therefore obvious that there exists an intrinsic relation between the notion of a dynamic coalition and *SOA*, with the former being a notion of the general collaboration of agents in dynamic environments and the second being a technical approach to how to support the former. However, *SOAs* itself may be dynamic coalitions, which leads to the following two views of the relation between *SOA* and dynamic coalitions.

1. On the software architecture level *SOA* may be seen as dynamic coalitions of services itself, which form according to the need of the larger software application. The *SOA*-composition or coalitions are dynamic in the sense that they are loosely coupled and dynamically bound at runtime. A service is an agent and a service composition is a dynamic coalition. This view of the relation between *SOA* and dynamic coalitions is a software engineering view dealing with the patterns according to which software applications are developed.

2. On a more abstract level of interaction between individuals, *SOA* may be seen as the means to implement dynamic coalitions on a software level. In this sense, *SOA* is solely the tool which is used in order to implement a dynamic coalition process on an architectural level, e.g. the method through which communication, interaction and information sharing takes place. The second view takes a more general stance and does not implicitly consider the dynamic coalition to be of technical nature only. For example agents may be human individuals, institutions or else, collaborating and *SOA* a tool to support dynamic coalition or parts of it.

It is clear to see that the second view is more general, i.e. that it also captures the first view. Since this thesis constitutes an abstract and general approach to the definition of a dynamic coalition, this second view is used throughout the work: Rather than to investigate the occurrence of dynamic coalitions in *service-oriented architectures*, this work investigates dynamic coalitions as a phenomenon completely independent of any software implementation details.[2] The reader may keep in mind that although this framework is abstract it is meant to serve as a tool for implementation through fitting software technologies, such as *SOA* and alike.

1.2 Solution Approaches

In this section related work is presented to show how the particular approach of this thesis differs from existing approaches

[2]From a general view this kind of approach is particularly important as dynamic coalitions may be implemented not only through *SOA* but through a number of technologies, such as *Cloud* or *component-based architecture*.

1.2.1 Related Work

The work at hand aims to integrate formal models of dynamic coalitions with access control policy models. In both fields a number of formalization attempts are to be found in literature.

Work on Dynamic Coalitions

Informal modeling of dynamic coalition (also known as *virtual organization*) has been subject to research for many years. Most approaches deal with the modeling of different aspects of dynamic coalitions, such as the designing and implementation process [Kat98, Win05], characteristics, structure and features of dynamic coalitions [LZK05, SLW98, RMR11] or behavior of dynamic coalition agents [SA13]. Intensive research efforts have been presented in the *GRID* field, from which the term "virtual organization" initially emerged [SL05, ZQG04]. Other approaches present informal models from the view of economic sciences [Klu98, Let01].

While all these approaches have been considered during the state of the art examination of this thesis and therefore also gave valuable input for the understanding of dynamic coalition aspects, eventually they turned out too specific or too focused on single aspects or implementation details. Furthermore, none of these approaches make use of any formalism and are therefore not suitable to achieve the goal of a formal modeling framework for dynamic coalitions.

In recent years several formalizations of single aspects of dynamic coalitions have been presented. Haidar et al. propose a formal model for PKI-based authentication in dynamic coalitions on the basis of a process calculus and the formal description language *Z* [HCA+09]. Bocchi et al. present formal description approaches for breeding environments in virtual organizations, which may be notably relevant in grid-computing [BFRRM09]. Nami et al. formalize dynamic coalition creation with the *RAISE* specification language [NSM07]. Esparcia and Argente formalize virtual organization structure according to the human organization theory [EA13]. Zuzek et al. formally model the coalition formation process through formalizing contract negotiation between collaborating parties [ZTS+08].

Although formal, all these approaches still focus on too specific aspects of dynamic coalitions. The actual paradigm of a coalition is only seldom defined let alone formalized.

A first formal approach to a holistic grasp of dynamic coalitions as a formal paradigm is to be found in the works of Bryan et al. who propose various models in the specification language *VDM* each covering what they call a "dimension" of modeling interest, such as *coalition membership*, *information transfer* or *trust* [BFJM06, BFG+08, BFJ+06]. A lot of the underlying work in this thesis is motivated by those works and certain aspects of the model structures have been adopted from [BFJ+06]. However, these approaches only formalize the structural aspect of the coalitions. Process properties have to be simulated through means of external tools (for example java-based) [BFG+08] to modify a state that is based on a *VDM*-model structure. This work extends this approach in so far as that it uses a formal method which allows for both, the structural modeling of dynamic coalitions as proposed by Bryans et al. as well as a formalization of state transition and thereby a formal way to define, test and simulate processes of dynamic coalitions in the same single formalism.

Only few other holistic modeling approaches for dynamic coalition modeling exist. The work most alike to the one presented in this thesis can be found in [MST09]. This framework catches formal descriptions of agents, services, roles and work flows. The used modeling techniques however are not based on well-known standards. Tool-support in creation and evaluation of dynamic coalitions design

as proposed by us, is therefore only hardly imaginable. Koshutanski et al. model highly dynamic coalitions, a subclass of dynamic coalitions, defined by them as coalitions with an extremely short life time [KM10]. They also do not make use of any modeling standard, which might create problems in critical application fields like crisis management or the health sector. Furthermore their model assumes a central management of the coalition in form of a coalition platform. Both limitations are too restrictive for the application scenarios that this thesis focuses on.

Work on Privacy & Access Control

Access control and privacy in information systems has been investigated since decades, therefore an exhaustive presentation of all literature to be found would be beyond this work. However, certain access control standards are subject to this work for which some literature will be mentioned here. [3]

- Identity-Based & Attribute-Based Access Control in *XACML*

  One part of this work partly formalizes *identity-based access control* or *attribute-based access control* on the basis of the OASIS standard *XACML*. In literature many attempts for such a formalization may be found. The one that influenced this work the most was [BFP06]. Independent from their previous work on dynamic coalitions a *VDM* formalization for *XACML* access control policies has been proposed. In this work a first suggestion to integrate the dynamic coalitions models with the *XACML* model has been suggested. This work takes up on this suggestion. Another more recent work from Masi et al. provides a *XACML* formalization which allows not only for *identity-based access control* but also for *attribute-based access control* policy definition such as the OASIS standard provides [MPT12]. They first propose a BNF-like grammar for the structure of policies and then provide a formal denotational semantics for this structure. Masi et al. work has also influenced the parts of this model which deal with *attribute-based access control*. However, as Bryans et al. they do not provide a formal way of defining actual access processes. Beyond that, a vast number of formalization approaches may be found in literature [KHP07, Bry05, Hoa78]. However, none of the suggested approaches take a similar approach to formalize dynamic coalitions, *XACML* policies and the processes over them in the same formalism.

- Role-Based Access Control

  As with *XACML* the formalization of *RBAC* access control policies has been extensively studied. [ZCW+07] use a notation called "Behavior" for specifying both the *RBAC* policies as well as the system design, which allows for safety verification of the proposed policies. [KMPP] use graph transformation for the formalization of the policies also allowing for tool-supported verification of the policies. [AK06] propose an approach which may be seen very close to the underlying work: They formalize *flat RBAC* in the specification notation *Z*, by the means of which they may also formalize an abstract state of a scenario and thereby test, simulate and verify. However, they do not integrate the other *RBAC* models such as *hierarchical RBAC* and separation of duty. Furthermore, neither of the mentioned nor any of the other approaches that were found during this related work investigation deal with the application of *RBAC* in dynamic coalition scenarios. Therefore, the proposed model of this thesis sticks very close to the *RBAC* NIST standard [SFK00], and proposes a new formalization in *ASM* which allows for the integration with dynamic

[3] For a detailed description of the various access control concepts discussed in this thesis, refer to section 2.2

coalition models.Therefore, a detailed description of this standard will follow in the preliminary part of this work.

- Trust-Based Access Control

  Trust in distributed systems and dynamic coalitions is a widely investigated field itself. However, most trust research is concerned with the way trust values are evaluated, i.e. with the trust evaluation functions and the parameters used in this computation [TB06, SHYL06, CGvH$^+$12]. The work at hand does not deal with such research questions. It is rather dedicated to applying the trust concept in an access control mechanism, called *TBAC*. This concept itself is also not new [CR06, LWR09, AMCGR05]. However, although applied in another field, the main question of these works continues to be how to evaluate trust in this new application domain. This thesis however, aims at making dynamic coalitions processes with *TBAC* formally understandable and verifiable. Therefore the actual evaluation, as well as the nature of the input parameters needed for the same, will be disregarded in this work. The resulting formal model supposes that trust has already been evaluated and the proposed *trust-based access control* infrastructure can be seen as a generalization or abstraction of many of the available *TBAC* models. Since the proposed model does not make any statement on how the trust values are to be evaluated, it allows for the later refining of the model by defining how the trust values are actually created, according to the modeler's need and the selected evaluation approach.

1.2.2 This Approach

This thesis is unique in the sense that it aims at integrating the notion of dynamic coalitions with a number of access control mechanisms as well as with a formal transition logic which allows for both the structural and process-wise formalization of privacy-sensitive dynamic coalitions in one single formalism: *ASM*. The structure of an *ASM* specification itself is congruent to the former mentioned needs: It consists of an algebraic structure which defines structural properties, such as universes and functions as well as a transition logic which allows to create programs which modify state over the algebraic structure, thus defining algorithms or behavior (see fig. 1.1). Several other higher-level state transition formalism offer similar features: the great flexibility and openness of *ASM*, its ability to model and simulate on any given abstract algebraic structure, as well as its to computer scientists natural way of programming behavior and algorithms, make it the first choice of this work.

The framework presented in this thesis will consist of a number of models, each one based on a first dynamic coalition model which defines the subject at hand without any access control mechanism. This basic model will then be extended to accommodate access control mechanisms, resulting in a new model for each access control mechanism.This approach to model dynamic coalitions as well as the access-control components in one single formalism, which allows for both the modeling of structure as well as processes in a formal way, is to the author's current knowledge unique. The usage of the well-researched *ASM*-formalism makes this approach more attractive because of analyzing techniques and tools that already exist.

Contributions of this Thesis

1. Informal, as well as formal, definition of the notion "dynamic coalition" in general, with different degrees of membership dynamics and with the required components and mechanisms for a privacy-sensitive dynamic coalition in particular.

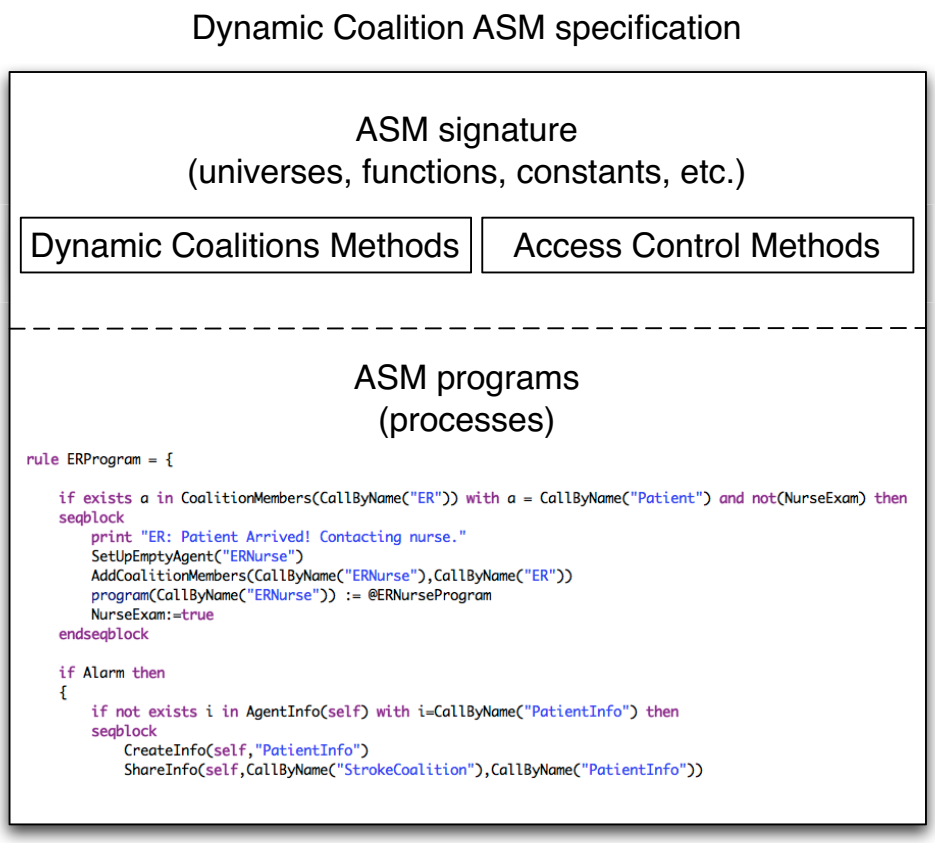

Figure 1.1: Simplified extract of an example *ASM*-specification.

2. Creation of a formal modeling framework, through means of the abstract state machine formalism.
3. Integrations of models for dynamic coalition structure with access control mechanisms for *identity-based access control (*IBAC*)*, *role-based access control* (*RBAC*), *attribute-based access control* (*ABAC*) and *trust-based access control* (*TBAC*)
4. Extension of the structural depiction by modeling of processes through means of *ASM* programs and runs.
5. Formal *ASM* rules for the administration of access control policies in dynamic coalition scenarios.
6. Demonstrations of the applicability of the proposed modeling framework by means of medical case studies taken from the case study pool of the *SOAMED Research Training Group* with which this thesis is affiliated.

Further Research Outcomes

- *State Exploration GUI* - Tool for the visual examination and demonstration of *ASM* simulations runs. In particular the interactions and developments of dynamic coalitions, their agents, information and policies are visualized and their temporal evolution may be investigated in order to facilitate validation.

- Demonstration of the interdisciplinary applicability of the model through integration of medical domain experts into the case studies.

Tools and Techniques used in this Thesis

- *CoreASM* for *ASM* signature definitions as a basis for the structural definition of dynamic coalitions and their access control policies as well as *ASM* programs for the integration of process modeling.
- *CoreASM* as a modeling and simulation environment.
- Self-made Java-Tools for a ASM *State Exploration GUI*

Publications of this Thesis

Through the course of the creation of this thesis a number of publications have been submitted by its author, sometimes in cooperation with associated researchers. These will be enlisted here in chronological order:

- S. Bab and N. Sarrouh, "*Formale Modellierung von Access-Control- Policies in dynamischen Koalitionen*" [BS11a]: In this paper the first research ideas towards a formal model for privacy-sensitive dynamic coalitions have been presented with the formalization approach still being based on *VDM* as proposed by Bryans et al. The main contribution in this paper has been conducted by the author of this thesis, while the associated researcher assisted in providing formal understanding for the outlook of the paper, which presented a discussion as to whether *ASM* or *Petri-Nets* are suited to provide a modeling technique, which (other than VDM) does not solely model structures of dynamic coalitions but also formalizes their processes. This paper may be seen as a recapitulation of the initial research motives and approaches.
- S. Bab and N. Sarrouh, "*Towards a formal model of privacy-sensitive dynamic coalitions*" [BS11b]: Building upon previous work of [BS11a] this paper went on in producing more sophisticated models of the previously proposed *VDM*-models for dynamic coalitions. However, it is shown, how the *VDM*-models may serve as the structural basis of an *ASM*-model, which allows for the formalization of processes. While the author of this thesis is the main contributor of the ideas presented in this paper, the associated author assisted in formulating first logical properties which were to be verified by means of the formal *ASM*-tools.
- N. Sarrouh, "*Formal modeling of trust-based access control in dynamic coalitions*" [Sar13]: This paper presents an extract of the thesis at hand. The focus of this work lies in the possible formalization approaches of trust aspects in dynamic coalitions and namely introduces the *Dynamic Coalitions with TBAC*-model of this thesis, presented in section 3.6. Excerpts from the corresponding case study (see section 4.2) were also presented in this thesis, as this paper presents a rather late stage of the thesis.

1.3 Thesis of this Work

The main thesis of this work is that the proposed modeling framework for privacy-sensitive dynamic coalitions supports the understanding and the development of software for these dynamic coalitions throughout the typical software engineering life cycle (*requirement capture*, *de-*

tailed design, *validation* and *documentation*): *Requirement capture* becomes possible through providing a framework for initial ground models for dynamic coalitions. A more *detailed design* of coalitions processes, as well as privacy aspects, may be modeled through the integration of various access control mechanisms into the framework as well as through the refinement of the process models. Because of the nature of the formalism used for this framework, simulation capabilities will provide means for the *validation* phase and the unique and simple nature of this formalism allows to use it as a *documentation* tool throughout the development process of supporting software for dynamic coalitions.

Note that one of Börger et al.'s phases of the software engineering life cycle has been left out in this thesis: "verification". This is not due to the fact that verification is impossible in the proposed framework. On the contrary it will be shown, that through the use of existing model checkers the model may be adapted such as that logical properties may be verified automatically. However, the necessary steps in order to make existing model checkers applicable to the proposed modeling framework are mainly software-technical issues concerning the code of the model checker and therefore do not fall into the focus of the subject of this thesis. Therefore, the thesis of this work leaves out the *verification*-phase. Nevertheless it will be shown how this particular feature may be adopted in future work.

The correctness of this thesis will be demonstrated on two case studies in the medical field. Furthermore, the validation of the framework as a whole is also conducted through the validation of the models presented in these case studies: As the processes in the case studies constitute dynamic coalitions and the presented framework is shown to be adequate to define, model and simulate these processes, it will be clear that the dynamic coalition framework itself is correct.

1.4 Outline of this Work

This thesis is organized as follows. After the first introductory chapter the second chapter will outline the preliminaries needed to understand the core of this model. These are:

- An informal definition of dynamic coalitions as well a categorization of dynamic coalitions according to the degree of their membership dynamics.
- An introduction to access control in general as well as to all access control mechanisms used for the models in this thesis: *identity-based access control (*IBAC*), role-based access control (*RBAC*), attribute-based access control (ABAC),* trust-based access control *(*TBAC*).*
- An introduction to *VDM* on which the models or Bryans et al. are based. These models serve as the initial motivation of this work and are therefore explicitly introduced.
- An introduction to basic as well as to distributed *Abstract State Machines* in a formal manner.
- An introduction to *CoreASM* a tool for the creation and simulating of *ASM* specifications. The models of this thesis make use of the *CoreASM*-syntax for the definition of *ASMs*.

The preliminary part is followed by the core chapter of this work: " A Formal Framework for Privacy-Sensitive Dynamic Coalitions". In this chapter the several models of which the framework is comprised will be defined and explained, namely:

- A basic dynamic coalition model in *ASM*, basically adapting some of Bryans et al. models into *ASM* and extending them with process abilities to allow for the sharing of information. This model will serve as the basis for all the models of the access control dimension, which are introduced in the following sections.
- A dynamic coalition model for *identity-based access control (*IBAC*)* enforcing privacy in the information sharing process. This access control mechanism will be identified as being suitable for dynamic coalitions with low membership dynamics.
- A dynamic coalition model for *role-based access control* (*RBAC*) enforcing privacy information sharing. This access control mechanism is identified as being suitable for dynamic coalitions with medium membership dynamics. The model here consists of several models, each one implementing the various *RBAC* concepts, such as *flat RBAC*, *hierarchical RBAC*, *RBAC* with static separation of duty, *RBAC* with dynamic separation of duty.
- A dynamic coalition model for *attribute-based access control* (*ABAC*) enforcing privacy for information sharing. This access control mechanism is identified as being suitable for dynamic coalitions with high membership dynamics.
- A dynamic coalition model for *trust-based access control* (*TBAC*) enforcing privacy for information sharing. This access control mechanism is identified as being suitable for dynamic coalitions with high membership dynamics as well.

The definition of the models is followed by an example *ASM* program for a fictional dynamic coalition scenario as well as a demonstration of the simulation capabilities of the *CoreASM*-environment.

In the fourth chapter of this work two case studies are presented. The thesis presented in the first chapter of this work will be validated against those two real-life scenarios. Both case studies are taken form the medical sector and have been created in joint work with institutions at the *Charité Berlin* which are cooperating with the *SOAMED Research Training Group* which this thesis is associated with. Both case studies constitute scenarios which are to be seen as dynamic coalitions of different membership dynamics with acute need of privacy enforcing mechanisms. As discussed in this first chapter the proposed models may be used at a number of stages of the software development life cycle. Thus, the presented case studies will address the different use cases of the framework: (*Requirement capture*, *Detailed Design*, *Validation*, *Documentation*) demonstrating the correctness of the thesis outlined in this chapter.

Chapter 2

Preliminaries

In this preliminary chapter an overview over the main topics, subjects and tools of this dissertation is given. In order to gain a first idea of the general subject of investigation, this chapter will start with an informal introduction to dynamic coalitions. In order to improve the understanding of these coalitions a basic categorization according to their degree of membership dynamics, as well as examples for each defined degree, will be presented.

This work deals with a certain type of dynamic coalitions, namely privacy-sensitive dynamic coalitions, meaning coalitions in which the information which is to be shared is highly sensitive and access to it has to be controlled. Therefore, brief introductions to common access control mechanisms and their functionality through which privacy in dynamic coalition scenarios may be enforced will be given in the second section of this chapter.

As mentioned in the previous chapter, first formal models for dynamic coalitions have already been presented in literature. The approach at hand starts from one particular modeling framework by Bryans et al. which is formulated by means of the *Vienna Development Method (*VDM*)* specification language [BF08, BFJM06, Bry05, BFP06, BFG+08, BFJ+06, BF07] In order to understand these first models a brief introduction to the basics of *VDM*, as well as to the specific models of dynamic coalitions in *VDM*, will be presented in section 2.3.

The main motivation of forming dynamic coalition is to cooperate in a common process by sharing information and resources. Therefore, the definition of processes is just as important as the ability to model dynamic coalition structures. This thesis takes up on the main ideas of Bryans et al. and seeks to extend the *VDM* models with the possibility to formalize processes and behavior. In order to reach this goal Bryans et al. had to use external applications (e.g. Java-programs) to modify the *VDM* structure instances. Different from this approach the work at hand seeks a unified formal representation of both the dynamic coalition structure as well as their processes in one formalism. To this end *Abstract State Machines (ASM)* were used. *ASMs* are not only able to specify the basic structure in a similar manner as Bryans et al. did by means of *VDM*, but are also capable of formally defining processes on these structures and thereby simulating and testing certain scenarios in a formal way. Because the structural definition and the process specification takes place in the same formalism a consistency of the formal definition of the two may be guaranteed and thereby contribute a model with allows for simulation, testing and even verification of dynamic coalitions from a structural as well as a behavioral perspective. Therefore, the basic concepts of *ASMs* in general, as well as in their distributed form, will be elaborated in the final section of this preliminary chapter. It will then be discussed in more detail why *ASM* is the suitable formalism to model dynamic coalitions and explained how supporting tools may be used for validation

and verification.

2.1 Dynamic Coalitions

The term “dynamic coalition”, as well as its synonyms, such as “virtual organizations” or “temporal alliance”, can be found in a wide field of research and software development projects for distributed systems (see section 1.2.1). However, the term itself is often used without proper definition, let alone a formal description of what a dynamic coalition is and how it operates. The lack of a formal definition of this kind is the exact motivation of the underlying work, which aims at providing a formal framework for dynamic coalition and thereby formally (by means of mathematically-based modeling languages) identifying and defining what a dynamic coalition is.

Since the core of this work deals with the formal definition of structure and operators of such coalitions, an informal (textual-based) definition of dynamic coalitions in this section will help to gain a first idea of the problem field. Furthermore a categorization according to the degree of the coalition's membership dynamics will be proposed. It is later (see chapter 4) needed to identify the character of dynamic coalition scenarios and also to gain important information for the decision making process on which access control mechanism is used in order to enforce privacy.

2.1.1 Informal Definition

Considering available applications for dynamic coalition as well as recent formalization efforts the following informal definition is proposed:

> A dynamic coalition is a temporary collaboration of autonomous agents through means of networking technologies, who share information or resources with each other, driven by the desire or the need to cooperate in order to achieve a common short-term, middle-term or long-term goal or to respond to an acute need. Agents may be individuals, organizations, programs, or coalitions of agents themselves. The attribute “dynamic”, describes the dynamic nature of the membership in such coalitions, in which new agents may join, others may leave and rejoin during the coalition runtime. The degree of the membership dynamics may vary.

It is obvious, that this definition does not pose any requirements considering the underlying infrastructure, such as a *SOA*, *Cloud* or *Grid* architecture. It is rather to be seen as a definition of the abstract (in the sense of abstracting from implementation details) concept of dynamic coalitions and their common characteristics. Dynamic coalitions as an abstract idea as given by this definition, can be found in various forms. The underlying architectures and infrastructures, the scale, the complexity, and the operation conditions, constraints and requirements for collaboration vary widely from one coalition to the other. However, these parameters are not commonly characteristic of the concept of a dynamic coalition itself, but are orthogonal problem fields, mostly addressed during the implementation of a dynamic coalition project.

2.1.2 Degrees of Membership Dynamics

Dynamic coalitions as defined in section 2.1.1 may be characterized according to the degree of their membership dynamics. It is important to mention that either of the following categories is excluding any of the other two, i.e. the nature of a coalition's

membership dynamics might be a mixture of different degrees. A certain part of the coalition could have a very low degree of membership dynamics, while another part is operating in a highly dynamic environment. Therefore, these degrees are providing merely a description framework of the dynamics in coalitions in order to have a common language in which to reason about them.

Low Membership Dynamics

In coalitions with low membership dynamics the set of collaborating agents is to the greatest extend predefined. Joining and leaving of agents into or from the coalition are very rare and because the coalition was set up to operate in this rather static membership environment the joining and leaving require a high amount of administration, in order to ensure the functionality of the collaboration. These coalitions may also be called *alliances*, meaning a collaborating set of agents, which rigorously define processes for joining and leaving of agents and the administration of these membership changes.

Medium Membership Dynamics

In coalitions with medium membership dynamics there exists a rather static set of predefined collaborating agents, but a dynamic change of this set by joining and leaving agents is foreseen and integrated into their operations in order to minimize the administration efforts for such changes. Centralized administration policies could include constraints or requirements on who is to join or leave the collaboration and how.

High Membership Dynamics

In coalitions with high membership dynamics very little to nothing is predictable about the set of collaborating agents. There might be a small group of coalition initiators, however the coalition aims at supporting the collaboration with mostly previously unknown agents. Administrative efforts are minimized as much as possible as a high flexibility of membership in the coalition is essential for the operation of the coalition.

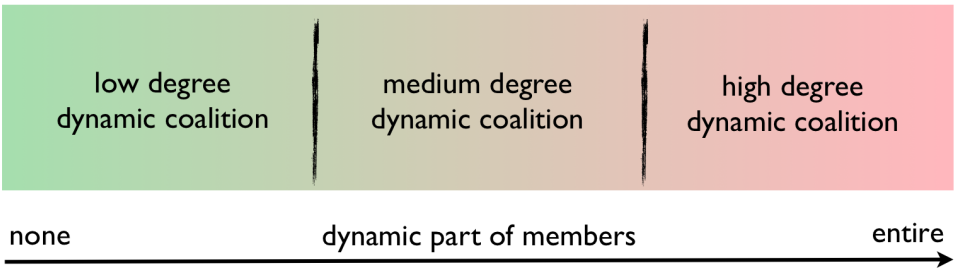

Figure 2.1: Classification of dynamic coalitions by membership dynamics.

Mixed Membership Dynamics

Coalitions may consist of clusters of collaborating agents with different membership degrees. For example a coalitions in which one part of the members are statically interacting while another part of the membership is highly dynamic, may be seen as coalitions with mixed membership dynamics as intuitively they can be seen as a mixture of coalitions with low and high membership dynamics respectively. However,

one could argue that the definition of a mixed membership degree is not necessary because a case like this could be regarded as a coalition of two sub-coalitions with different membership dynamics. Therefore, this thesis will omit this particular degree of mixed membership dynamics in the enlistment of dynamic coalition examples in the following sections. However, conceivable examples may be easily constructed through combination of examples of the other coalitions types.

2.1.3 Examples of Dynamic Coalitions

Examples of dynamic coalitions can be found in various fields. Here a few will be named and ordered by the degree of their dynamics.

Low Membership Dynamics

Examples of coalitions with low membership dynamics are:

Hospital scenario: In a hospital the agents which collaborate are predefined according to the job description (doctor, nurse, etc.). Therefore, it can be seen as a coalition with low membership dynamics, with a relatively high administration effort of joining and leaving of new employees. The goal of this coalition could be formulated as follows: providing health care for the sick or the injured. It will be shown that there also might be coalitions inside a hospital, which are of medium or even of high membership dynamics.

Military alliance scenario: In a military alliance scenario, such as NATO, the agents (in this case the allied nations) which collaborate are rigorously pre-defined and the joining and leaving of agents, can only be achieved by following complex membership administration policies.

Medium Membership Dynamics

Examples of coalitions with medium membership dynamics are:

Hospital Scenario: From the perspective of individual patient treatment coalitions with medium membership dynamics may also be found in a hospital. Certain legal constraints have still to be met, such as “At least one doctor has to supervise the patient treatment.” and so on but which doctor, institute, diagnostic measure or treatment has to take place is individually defined by the needs of the patient and is therefore only to a very limited degree definable. Patient treatment may also need unforeseen collaborations across the borders of the department or even across hospital borders, when communicating with external doctors, experts, etc. Thus, each patient treatment process can be seen as an instance of a dynamic coalition with medium membership dynamics, which forms according to the needs of the patient.

Crisis management: In crisis management coalitions with certain predefined agents form (for example police and government institutions during a terror attack or a pandemic, which always have to be part of the crisis solution collaboration), but the coalition has to remain flexible enough to deal with unforeseen collaborations with agents that are needed in order to overcome certain aspect of unforeseen or even unforeseeable aspects of the crises.

High Membership Dynamics

Examples of coalitions with high membership dynamics are:

Online Social Networks Initially *online social networks* are basically empty of any agents, and provide only the means of joining, leaving with the goal of interacting with other agents. These changes in membership are an essential part of the nature of the network. Very little to no administration efforts should be necessary

to manage membership changes. It may therefore be seen as a coalition with high membership dynamics, although it has to be said that inside the coalition of the online social network other coalitions with different degrees of membership dynamics may form, for example in individual friend lists, circles or groups. One example of an *online social network* will be closely investigated in the course of this work: a research data exchange platform for researchers in the medical field, in which research data is shared with other, possibly unknown researchers.

Multi-agent systems: Multi-agent systems in which program-agents are designed to use input from program-agents around them such as in mobile phone networks or in wireless sensor networks are examples of coalitions with high membership dynamics. Coalitions of sending, transmitting and receiving agents are constantly created and dissolved as the underlying topology of the network changes, for example when new mobile phones come into range of a certain radio network or sensors are activated or deactivated due to change of their environmental parameters.

2.2 Access Control Mechanisms

Access control mechanisms as tools to enforce privacy of information and resources have been extensively investigated since the first technological possibilities of information sharing. Various types of access control have been defined, such as *identification-based access control* or *role-based access control*. A number of national and international institutions such as *United States Government Department of Defense (DoD)* or the *Organization for the Advancement of Structured Information Standards (OASIS)* have since developed standards, defining the basic concepts and architecture elements for the different types of access controls.

Although a significantly higher number of access control mechanisms has been proposed in literature, such as *mandatory access control* [Lin06] or *credential-based access control* [Lee11, BFL96]. However, this work identifies four access control concepts in accordance with the *National Institute of Standards and Technology* as the main tools to provide the means to ensure privacy in the specific scenario of a dynamic coalition. These are: *identification-based access control*, *role-based access control*, *attribute-based access control* and *trust-based access control*. The decision to regard these as the main mechanisms to enforce privacy in dynamic coalitions, builds as will be shown in the following sections, upon their fields of application which hold similarities of the different degrees of the coalitions membership dynamics, which have been defined in section 2.1.2.

All these concepts define policies of different structures which define the different aspects at the core of the access control mechanism. They may therefore be called *policy-based* although the term *policy-based access control* is also used in literature for a generalized ABAC approach which unifies policies definitions across organizations and institutions. When referring *policies* in the following, this particular meaning is not meant, but instead the term is used to describe a certain set of rules and permissions with define the way in which access control is supposed to be carried out.

In the following sections a brief overview of each of these access control mechanisms will be provided along with their components, requirements and structures of their policies according to which the access is governed. For *identification-based access control*, *role-based access control* and *attribute-based access control* well-established standards define how access control policies may look like. *Trust-based access control* are relatively new concepts in relation to the aforementioned, therefore well-established standards are not available. Nevertheless, the common concepts and ideas in literature will be extracted in order to provide a basic definition on which the formal model presented in this work will be based on. For each mech-

anism it will be shown to which degree of membership dynamics it is suitable for.

2.2.1 Identification-Based Access Control (IBAC)

Identification-based access control means that access control decisions are determined through a user's identity. In literature *IBAC* is more often referred to as *discretionary access control (DAC)* and sometimes as *mandatory access control (MAC)*. However *DAC* states that the access control policies are administered by each user themselves, whereas *MAC* requires a centralized administration. The term *IBAC* abstracts from the specifics of administration and summarizes all access control mechanism which have identities as subjects for their access control decisions. In *Trusted Computer System Evaluation Criteria (Orange Book)*[1], a *DoD*-standard for assessing effectiveness of computer security, *discretionary access control (DAC)* is defined as to

> "[...]define and control access between named users and named objects [...]. The enforcement mechanism [...] shall allow users to specify and control sharing of those objects by named individuals or defined groups or both."[2]

In the following the term *identification-based access control (IBAC)* will be used exclusively, because it emphasizes the user's identity, which lies at the core of the access control evaluations. In *IBAC* an owner of an object defines a policy which states which other subjects , i.e. which other users have access to that object and what kind of access they may perform. The ownership of objects is particularly important in *IBAC*, because only the owner may define the access policies for his objects.

IBAC-Standard: OASIS XACML

The *eXtensible Access Control Markup Language (*XACML*)* is a well-established *OASIS*-standard based on *XML* for the description and definition of access control policies in distributed systems.[3] A simplified visualization of the functioning of *XACML* policies may be found in figure 2.2. Requests from subjects to access certain objects are intercepted by the so called policy enforcement point (PEP) and sent to the policy decision point (PDP) as a standardized *XACML* request. With access to the policies it is up to the PDP to check whether an access is to be permitted or to be denied. It sends this information to the PEP which is responsible for the granting or denying of access to the object respectively.

The policies may define which access requests are responded to with a "permit", a "deny" or a "not applicable" answer. The latter happens if an access request has not been matched with an answer in the policy. It is up to the designer how to deal with these cases. If access control is *deny-biased*, access control will only be granted if the response is "permit", else the access will be denied and vice verse if access control is *permit-biased*.

Due to the openness and extensibility of the XML-based *XACML*, policies of various access control concepts, such as *attribute-based access control* (*ABAC*) or *role-based access control* (*RBAC*) may be defined in *XACML*. However at the very core *XACML* provides a basic syntax for defining *identification-based (IBAC)* policies which define "permit" or "deny" answers for requests of *subject-object-action*-structure such as:

[1] See `http://www.boran.com/security/tcsec.html` for the full book.
[2] Taken from:`http://www.boran.com/security/tcsec.html#C`.
[3] See `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml` for the various versions of *XACML* specifications.

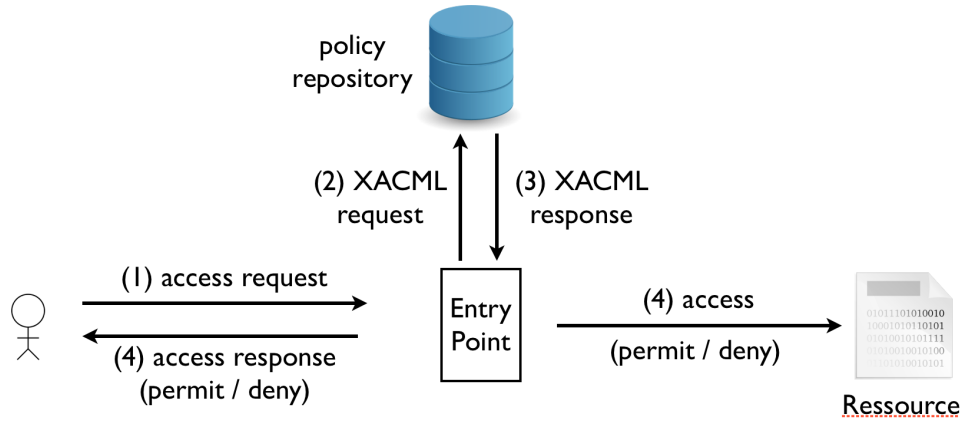

Figure 2.2: Simplified presentation of *XACML*-architecture according to the OASIS standard.

$$\{(\text{subject}, \textit{Alice}), (\text{object}, \textit{text.txt}), (\text{action}, \textit{read})\}$$

with the meaning of Alice wanting to access the text file by reading it. Since *XACML* provides templates for defining those policies, the proposed framework of this thesis will make use of those *XACML*-structures in order to define *IBAC*-policies. In a distributed system e.g. a dynamic coalition, each agent would have to be equipped with PDP,PEP, etc. and could then define policy rules which govern access to his own information. Sharing of data would then be governed by the use of *XACML* access requests and responses.

IBAC: Access Control for Coalitions with Low Membership Dynamics

It seems obvious that a fitting setting for the use of *IBAC* would be a dynamic coalition with rather low membership dynamics. Since owners of objects have to specifically define access rights for single subjects, the administration effort to maintain these policies is quite high, especially if every user defines those policies for himself, which makes changes difficult as every user would have to be instructed personally to change certain access control rights. As soon as new, previously unknown users join the coalition and want to gain access to objects their owner will manually have to implement new rules in order to make that access possible, or else the access will be denied or permitted for *permit-biased* respectively *deny-biased* access control. If the membership dynamics are low, e.g. the coalition membership fluctuation is minimal and the agents in the coalition are to a high degree well-known and predefined. Hence, the administration efforts for *IBAC*-policies will be manageable.

Therefore, *identification-cased access control (IBAC)* may be seen as an ideal method control access in dynamic coalitions with low membership dynamics.

2.2.2 Role-Based Access Control (RBAC)

In large-scale networks defining access control rights for each subject at a time can be impractical to impossible. As a solution to the huge administration efforts needed to maintain *identification-based access control* policies in such networks, the *National Institute of Standards and Technology (NIST)* provides a standard for a role-based security solution. *Role-based access control (RBAC)* has already been

formalized in 1992 by Ferraiolo and Kuhn [FK92] and in 2000 an extended set of models has been published as a NIST standard [SFK00] and was adopted as an ANSI/INCITS[4] standard in 2004.

The standard is comprised of a number of sub-models which may be seen as building on each other with every next model providing additional functionality for various application fields. The main models are:

Flat RBAC

> 'Flat *RBAC* captures the features of traditional group-based access control as implemented in operating systems [...]. The features of flat *RBAC* are obligatory for almost any form of *RBAC* and are almost obvious."[SFK00]

The core of the NIST standard which has already been proposed in [FK92] identifies access permissions to objects and assigns these permissions to roles, which for their part are associated with users (see figure 2.3). Through the association to a role a user may gain access permissions without the necessity of pre-defined unique access control rules for each user at a time.

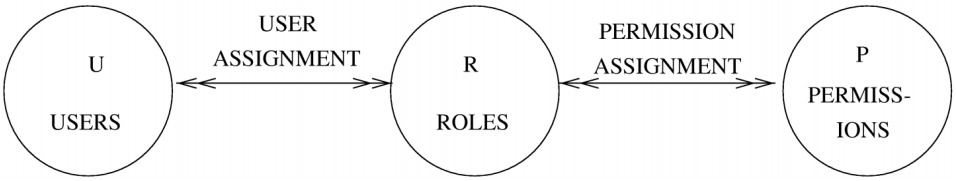

Figure 2.3: Visualization of *flat RBAC* model taken from [SFK00].

User assignment and *permission assignment* are many-to-many relations between users and roles or roles and permissions respectively: Users may act in various roles, whereas one role may be assigned to several users. Similarly a role can have several access permissions attached and a single permission can be given to various roles.

As soon as a user logs into an information system one, several or all of his assigned roles are activated. This role activation grants him the permissions assigned to the activated roles.

Hierarchical RBAC

Hierarchies in role definitions make way for inheritance of role permissions. Mathematically role hierarchies are partial orders on the set of roles, defining which higher (senior) role is to inherit the permissions of a lower (junior) role.

Hierarchies may be arbitrary partial orders in which case they are called general hierarchies. The NIST standard also recognizes that in certain fields restricted hierarchies are desirable, meaning hierarchies which impose restriction on the hierarchy structure, for example demanding that the hierarchy is of a tree structure.

Constrained RBAC

In *constrained RBAC* constraints on the user-role assignment, as well as the role activation, may be defined in order to implement requirements for *separation of duty (SOD)*. Separation of duty constraints are inherited in hierarchies but may also be applied in flat *RBAC* scenarios.

[4]See `http://www.ansi.org/` and `http://www.incits.org/`

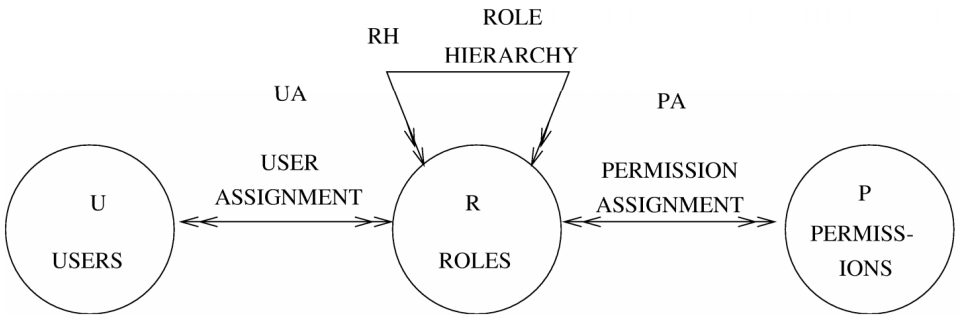

Figure 2.4: Visualization of *hierarchical RBAC* model taken from [SFK00].

Static separation of duty (SSOD) means imposing constraints on the assignments between user and roles, which define that if a user is authorized for a certain role, he is forbidden to be a member of a certain second role (see figure 2.5).

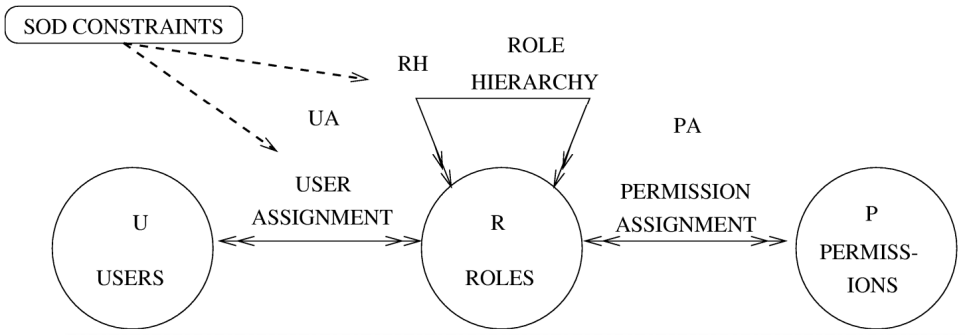

Figure 2.5: Visualization of *constrained RBAC* model with *static separation of duty (SSOD)* taken from [SFK00].

Dynamic separation of duty (DSOD) makes use of *sessions* in order to impose constraints on the actual activation of a role, meaning that a user may be assigned to a certain set of roles, which are not allowed to be activated at the same time. Thus, *DSOD* constraints do not only impose restrictions on the user-role assignments, but also on the possible combinations of activated roles in a session (see figure 2.6).

Administrative RBAC

Although not an actual part of the *RBAC NIST* standard, administrative *RBAC* has also been proposed by Sandhu et al. with the intention of using *RBAC* concepts themselves to administer and govern *RBAC*-policies [SBC+97]. In a similar way to the models mentioned in section 2.2.2, users that are supposed to administer role and permission assignment, will be assigned to so called administrative roles, which provide with administrative permissions e.g. the permission to change user-role-assignments, role-permission-assignments, role-hierarchies and constraints on either one of these (see fig. 2.7)

Administrative *RBAC* aims to facilitate the assignment process and therefore increase *RBAC* practicability in networks with a larger scale and particularly with higher membership dynamics. Since administrative RBAC makes use of the same concepts that RBAC proposes on a meta level, it will not be part of the models in this framework. The integration of it is straightforward and will be subject of design decisions. However, in the upcoming models, some considerations on who is

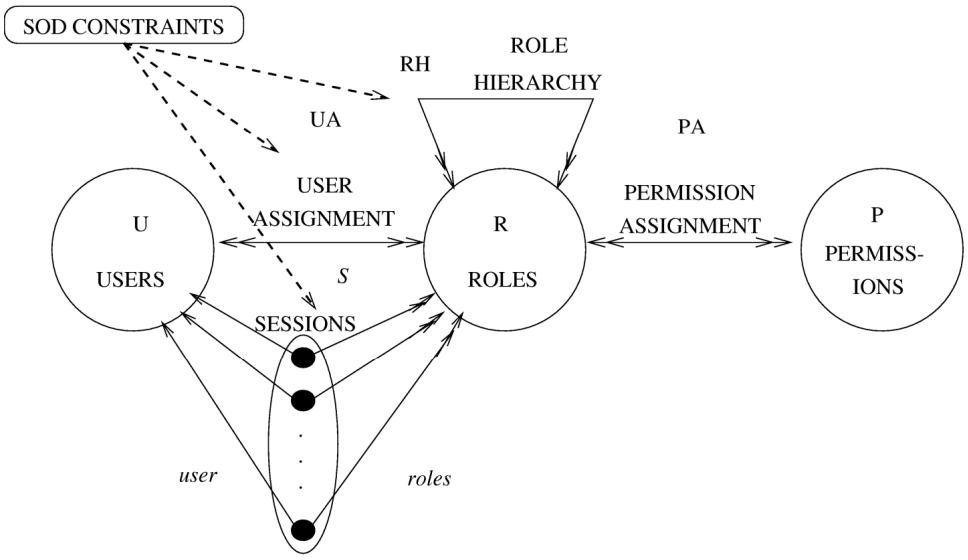

Figure 2.6: Visualization of *constrained RBAC* model with *dynamic separation of duty (DSOD)* taken from [SFK00].

to administer the role structures will be made. For more details see section 3.3.1.

RBAC: Access Control for Coalitions with Medium Membership Dynamics

The definition of roles and the assignment to users (in this case coalition agents) facilitates the creation of access control policies in comparison to the *identification-based access control* model. Access permissions of large numbers of agents may be relatively easily defined, changed or revoked, without the need to address every agent one by one. Hierarchies, constraints and separation of duty deliver desirable functionality to reduce complexity in the role assignment and express requirements to the underlying role structures.

An important question in networks in general and in dynamic coalitions in particular is, who may create, assign, revoke or delete roles, permissions, hierarchies and so on. In a dynamic coalition with medium membership dynamics a certain set of agents form a relatively static part of the coalition. However, dynamic changes of this set have to be foreseen in order to make collaboration with previously unknown agents manageable. It is reasonable to assume that a predefined, static part of the coalition may jointly take to these administrative tasks to manage the other part of rather fluctuating members. Who administers the assignment relations and how, can be easily defined by the administrative *RBAC* component.

> **Therefore, *role-based access control* in its various forms may be seen as an ideal method to control access in dynamic coalitions with medium membership dynamics.**

2.2.3 Attribute-Based Access Control

Attribute-based access control (ABAC) was first introduced in order to overcome obvious shortcomings of the role-based approach: First, In *RBAC* it is difficult to define granular access control for single users. In order to achieve *identification-based access control* rules, roles and sub-roles have to be created and composed,

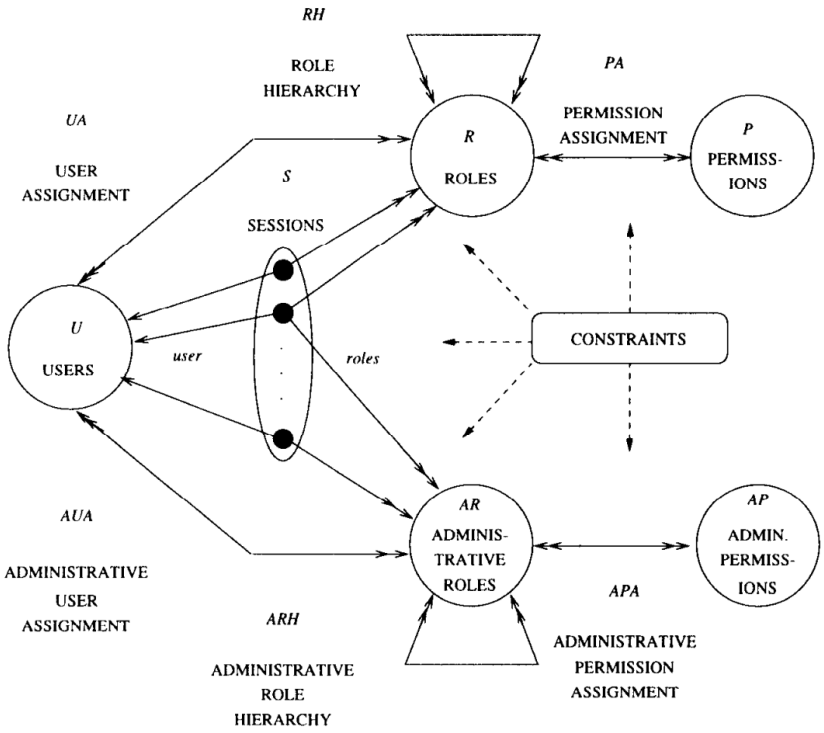

Figure 2.7: Visualization of *administrative RBAC* model taken from [SBC+97].

which makes it error-prone to secure that all users have the required rights, but not more than those. Secondly, roles, role hierarchies and permission assignments have to be defined through agreement with all partners and maintained by a single or group of administrators. Furthermore, *RBAC* provides only limited context consideration, such as time, location, or environmental state, which would provide help to capture dynamically changing access requirements in a dynamic coalition. Hence, as complexity and dynamics increase, the use of *RBAC* becomes more difficult and error-prone. In dynamic coalitions of high membership dynamics, a different way to address access control is necessary. Therefore ABAC was created to provide

> "[...] the ability to differentiate individual members of a group and to selectively allow or deny access based on a granular set of attributes." [oST09]

Hence, in ABAC the access control decisions are based on attributes, which may be taken not only from the requesting subject itself, but also from the environment or the requested resource. The attributes are distinct well-defined information fields, for example subject attributes, such as *role*, *employer*, etc., resource attributes like *type*, *size* etc. or environment attributes, such as *time*, *location*, etc.

In ABAC policies are not limited to the subject-object-action structure as proposed in *IBAC* (see section 2.2.2) but can be expressed with greater freedom. For example, policy rules that define access for the following request could be created:

> {(subject.role, *nurse*), (subject.time, *10:00*), (object.name, *text.txt*), (action.type, *read*), }

Policies are expressed in logical formulas. For example the following policy defines that a nurse may access the text.txt file in the time window between 10:00 am to 06:00 pm:

If (subject.role=*nurse*)
∧ (subject.time >= *10:00*)
∧ (subject.time <=*18:00*)
∧ (object.name=*text.txt*)
∧ (action.type=*read*)
then permit
else deny

Attributes may be pushed with the request or also be pulled by the policy decision point, i.e. the attributes do not have to come from a single attribute repository but may be retrieved from different information systems.

XACML not only serves as a a means for implementing *IBAC* policies but has nowadays also become the de-facto standard for defining ABAC-Policies. *XACML* defines certain XML-fields for defining specific attributes such as the following field for defining the role of a subject:

```
<SubjectAttributeDesignator
    AttributeID="urn:oasis:names:tc:xacml:2.0:subject.role"
    DataType="http://www.w3.org/2001/XMLSchema#string" />
```

Hence, as before in *IBAC*, the *XACML* policy structure is used as a template for the formal model in later sections.

ABAC: Access Control for Coalitions with High Membership Dynamics

ABAC allows for the creation of access control policies which are not limited to subject identities or roles but may take attributes of the requesting subject, the resource, the action and the environment into account. Indeed, it is not even necessary that users are known before they request information as long as the attributes needed for the pre-defined policy evaluation are provided or are accessible. Attributes from the environment, such as time, location or state of systems, may also help to reduce the complexity of the evolving dynamics.

Since identifiers and roles may also be represented by attributes, ABAC is powerful enough to express policies based on both, *identification-based access control* and *role-based access control* and may therefore be regarded as a superclass of the two.

However, ABAC has some shortcomings: Attributes have to be well-defined, meaning that users indeed do not have to be known in advance, however they need to provide the needed attributes in a pre-defined format in order to take part in the coalition. *Policy-based access control (PBAC)* tried to address this problem by harmonizing disparate attributes and access control mechanisms. However, *PBAC* proved to be difficult for policy definition and implementation [oST09]. Furthermore, attributes are static in the sense that real-time development of attributes may not be take into account , i.e. it is not possible to define access on the basis of a certain development curve of attribute changes. Hence, it is not possible to express *risk* and *trust* and base policy decisions on those values.

> **Thus, *attribute-based access control* may be seen as an ideal method to control access in dynamic coalitions with high-membership dynamics as long as disparate attributes in the coalition are harmonized and the dynamic evolution of attributes is not of interest**.

In coalitions with even higher dynamics, for example in unforeseen emergency scenarios the requirements for the applicability of *ABAC* might not be met. Therefore, in the next section the even more dynamic approach of enforcing access control on the basis of the notion of trust will be introduced.

2.2.4 Trust-Based Access Control (TBAC)

Various researchers have found the role-based access control model to be impractical for dynamic coalitions, in which membership is highly dynamic [CW13, oST09]. The agents which participate in coalitions with high membership dynamics are not previously known and due to the nature of the dynamics there are no ways to administer the access control policies in a centralized way such as *RBAC* proposes [BFL96, LM03].

Attribute-based access control has been introduced in order to address those issues. However in order to be applicable in dynamic coalitions a difficult and complex harmonization of the agents attributes is required. Furthermore, dynamic evolution of attributes has not been considered by ABAC. Hence, a more dynamic approach of enforcing access control is needed, which takes into account dynamic metrics of risk and trust. This approach may be called *trust-based access control*, although in literature the name *risk-adaptive access control (RAdAC)* may also be found [oST09]. This thesis makes use of the former as it bears a closer resemblance to the intuitive notion of trust, which lies at the very heart of access control decisions in this approach.

In order to overcome the above mentioned shortcomings for highly dynamic scenarios, the notion of trust has been proposed as a main criteria for determining access rights in highly dynamic coalitions [LWR09, AMCR04, AD05, CR06]. As a notion which originated in sociology, trust as a paradigm in distributed and dynamic information systems has been extensively researched. However, although various models have been proposed, there is no standard definition and only moderate consensus about what trust exactly is or is not.

The aim of my work is neither to contribute a unifying theory of trust nor add any research contribution in the trust field itself. It rather intents to utilize the basic and common concepts of trust research to establish a formalization of how *trust-based access control* can contribute to solving privacy problems in dynamic coalitions. In particular it will not define how trust is actually assessed or evaluated and what parameters are substantial for this calculation. These questions are seen as not in the scope of this work of this work and the various possibilities for answering them are left open for the later actual implementation.

The following sections will give a short overview over the notion of trust. Since the scope of this work does not allow an extensive cover of all the ways in which trust has been understood in literature, the presentation will be limited to the aspects that are identified as critical for the formal integration of trust-based access control into the proposed framework of privacy-sensitive dynamic coalitions of this thesis.

The Notion of Trust

> "[...]a means for reducing the complexity in society."[Luh00]

According to Luhmann, the above mentioned complexity is a result of interacting individuals with their own motivation and goals. Trust as a "particular level of the subjective probability with which an agent will perform a particular action" [Gam88] can be seen as the "foundation, or the dynamic precondition for any free enterprise society"[rcsff01]. Simply put: If agents want to collaborate in a dynamic environment, in which many agents are previously not well or not at all known, a

certain evaluation of their "trustworthiness" on the basis of trust indicators such as credentials, reputation, recommendation or interaction history and experience with particular agents, may help to determine whether an agent should be granted information access or not.

In computer science this "trustworthiness" of one agent is mostly determined through trust evaluation functions which take the above mentioned trust indicators of this agent as parameters. In most models these functions return a real trust number, a so called *trust degree* which symbolizes the level of the trust one has for a particular agent [LM03, BFL96]. Some trust models envision agents having trust lists which are updated as soon as new trust indicators are available or after a certain amount of time [CR06]. Other more recent conceptual or formal models refrain from the storage of pre-evaluated trust lists and concentrate on the policies which define the trust relations. These trust policies similar to the trust evaluation methods, take trust indicators at a certain time of a run of a system and then freshly evaluate the trust for every transaction at a time [EN10].

As the question of how and when to evaluate trust degrees has been extensively covered in literature and is also very context dependent in considering which parameters to use as trust indicators, this thesis refrains from modeling this evaluation process and assumes as in [BFJ+06] that the trust degrees have already been obtained for every agent. Another question in trust research deals with how the trust degrees are to be used the answer also being highly context-dependent. In the context of the second case study in this thesis it will be utilized for access control decisions.

Trust Values for Access Control

When trust degrees are used as the main criteria to decide if access request from agents should be granted or denied, so called *trust-based access control* (*TBAC*) takes place. As the information society underlies increasing degrees of dynamics, this access control aspect of trust has been investigated in recent years [AMCGR05, CR06, AMCR04]. After their evaluation trust degrees are the foundation for access control decisions, with access control policies defining which access permissions are associated with which trust degrees.

Figure 2.8 shows a similar structure as found in the *role-based access control* architecture (see figure 2.3 for the *RBAC* model). However, there are two important differences: On the one hand, figure 2.8 shows the local policy structure of one agent and not the global structure of agents, roles, permissions and their global assignments as in figure 2.3. On the other hand the relations between agents and trust degrees and between trust degrees and permission are many-to-one or one-to-many relations respectively. Thus, in *TBAC* policies an agent may assign, one and only one trust degree to each agent he knows, whereas one trust degree may be assigned to many agents. An agent may also assign one or several permissions to each trust degree and thereby defining what access control permission are granted to agents of which trust degrees. The actual evaluation of the trust degree takes place in the *trust evaluation policies* which takes static and dynamic attributes as parameters and provides the trust value according to an evaluation function which the agent has to define.

TBAC: Access Control for Coalitions with High Membership dynamics

Similarly to *ABAC*, *trust-based access control* aims at dynamic environments in which interacting users are not previously known. Those scenarios are regarded as dynamic coalitions with high membership dynamics. In these uncertain environments with little or no static member foundation the definition, maintenance or ad-

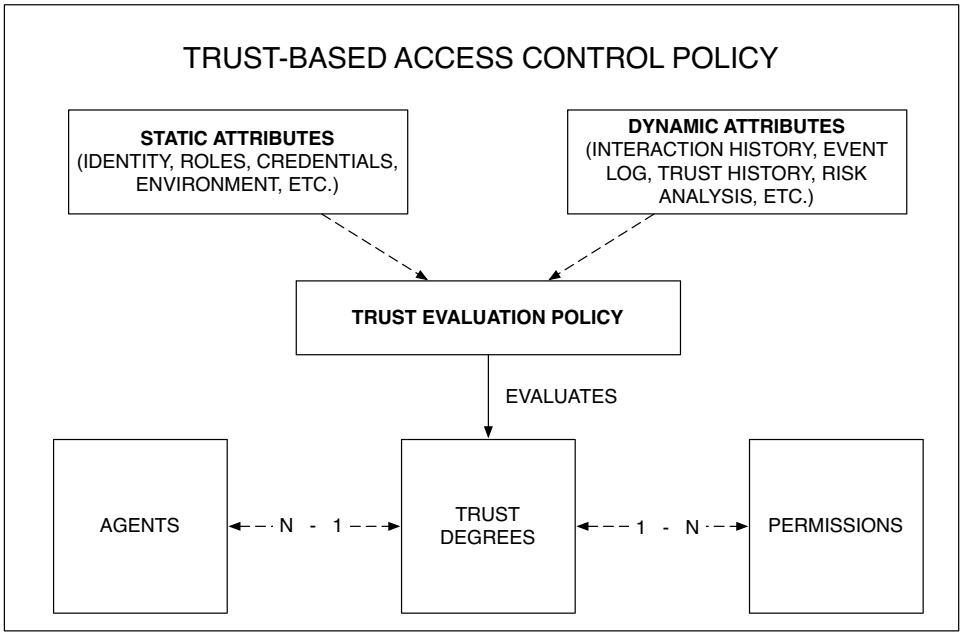

Figure 2.8: Exemple access control policy of one agent.

ministration of *IBAC* and even of *RBAC* policies may be impractical to impossible, because the environment on which the policies is based changes too rapidly. Trust as basis for evaluating access control decisions on the basis of trust indicators, such as agent attributes, credentials, reputation, recommendations or interaction history has been propagated as a fitting solution for these scenarios, reducing complexity and administration efforts to a minimum. Note that *TBAC* is also an extension of *ABAC*, i.e. *ABAC*-policies are expressible through *TBAC*-policies, with the extension that *TBAC*-policies allow for the consideration of not only static attributes (like in *IBAC*, *RBAC* and *ABAC*) but also dynamically evolving attributes, such as interaction history, event logs and more. All those attributes become parameters for the trust evaluation function, which basically serves as the policy and determines the trust degrees which each agent may assign to access permissions to his resources.

Thus, *trust-based access control* can be seen as an ideal method to control access in dynamic coalitions with high membership dynamics.

2.2.5 Summary: Dynamic Coalitions and Access Control

Historically, access control models underwent an evolution from coarse-grained *IBAC* policies over increasingly finer-grained *RBAC* and *ABAC* policies to *TBAC* policies, in order to deal with increasing dynamics of systems (see figure 2.9).

Coalitions with low membership dynamics offer a fitting environment for *IBAC*. *RBAC, ABAC* or *TBAC* may also be applied in those scenarios, however the benefit for deploying them is quite low as they are designed for rather dynamic or fully dynamic environments respectively. Coalitions with medium membership dynamics are an ideal field for applying *RBAC* because a rather static part of the coalition may handle the reasonable administration efforts for membership fluctuation. These efforts will get too high in a highly dynamic environment as the static foundation of members is minimal and membership changes are too frequent. Therefore *ABAC*

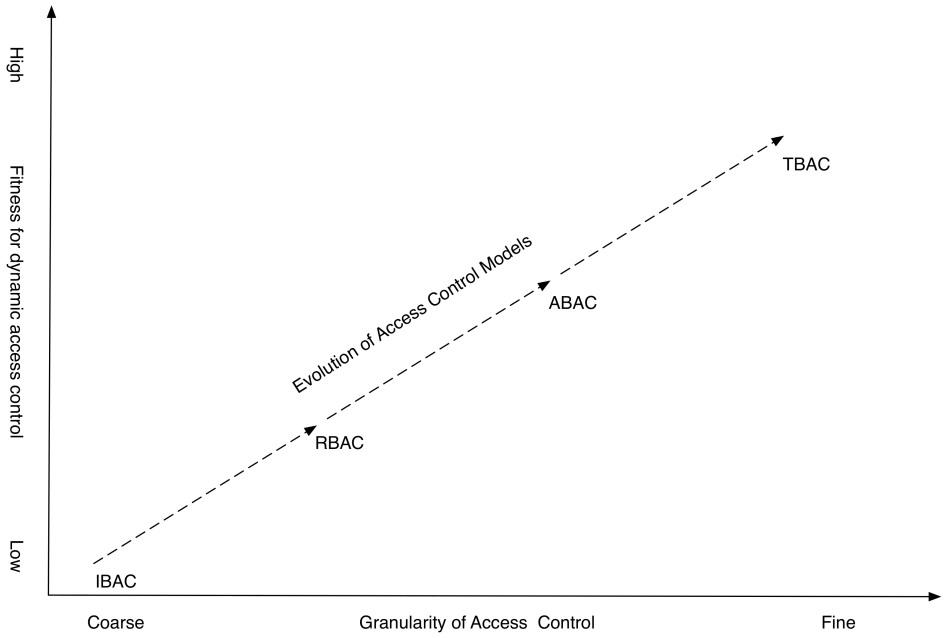

Figure 2.9: Access control granularity vs. fitness for dynamic access control on the basis of [oST09].

and *TBAC* may be seen as the ideal methods to deal with privacy in coalitions of high and higher membership dynamics respectively.

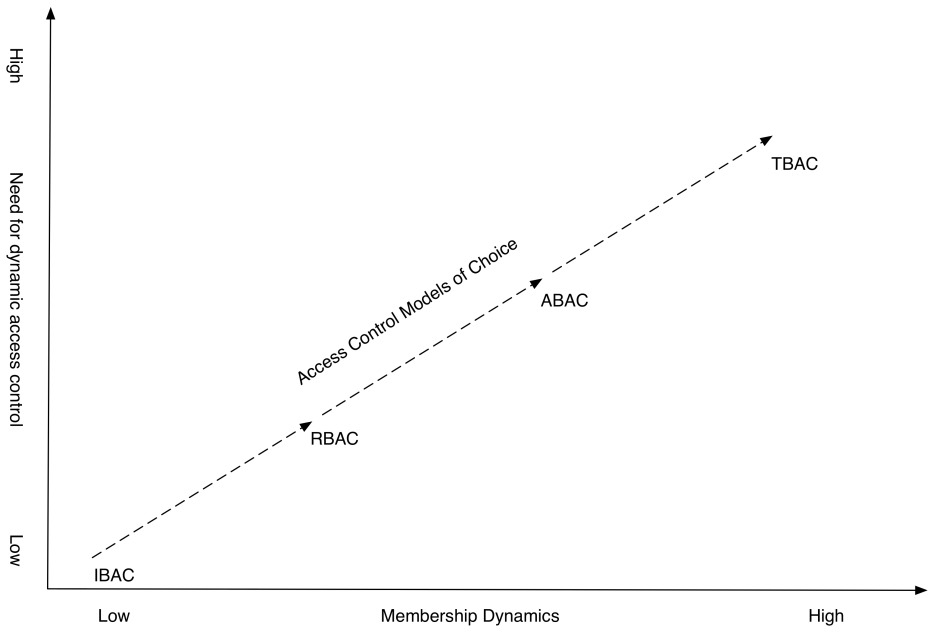

Figure 2.10: Access control vs. coalition membership dynamics.

2.3 The Vienna Development Method (VDM)

The Vienna Development Method is tool set of formal methods developed at IBM's Vienna Laboratory in the 1970. It has since been extensively used to model and specify computer-based systems. *VDM* is mainly identified through the *VDM* specification language (*VDM-SL*), but includes several tools for modeling, testing and proving of specifications as well as executable code generators.[5] An ISO-standard for the specification language is also available. In this thesis a particular dynamic coalition modeling approach of Bryans et al. which has made use of this formal method is of interest. Various case studies have demonstrated the applicability of this approach, which serves as a first step and inspiration for this dissertation. Thus, in the following sections an introduction to *VDM-SL* and its object-oriented extension *VDM++* is given. [6] As Bryans et al. the syntax from the ISO-definition will be used here and it will be refrained from a mathematical syntax, which may also be used to specify *VDM*-models. Finally the works on modeling dynamic coalitions proposed by Bryans et al. whose models will later serve as the main inspiration for modeling the algebraic structure of the *ASM* framework proposed in this thesis.

2.3.1 Basic Concepts of VDM-SL

VDM-SL models are structured like many programs and therefore can easily be read by most software developers. A *VDM* model is build from descriptions of data through type and value definitions as well as functions, which provide functionality over this data. In [FL09] an introducing example for models in *VDM-SL* may be found on the first pages, dealing with the specification of a simple air traffic control system.

Data can be described by using basic data types, such as *real*, *int*, *boolean*, etc. Consider the following data representation for the latitude of an airplane, which is needed to provide information about its position. Invariants may describe restrictions on the data types, which have always to be true for any values of this type. In this case the invariant defines latitude as being a real number between -90 and 90 as a representation for degrees:

```
Latitude = real
inv lat == lat >= -90 and lat <90
```

complex types may make use of such type definitions, such as the complex type for representing the actual position of an aircraft:

```
AircraftPosition :: lat   : Latitude
                    long  : Longitude
                    alt   : Altitude
```

More complex types may be created through the use of *sets*, *sequences* or *mappings* of values or in between different value collections respectively. As soon as a first type description has been done, functions may model the behavior of the system. A function definition consists of an input and output type definition as well as a definition of how to evaluate the output result of the function based on the input values. A pre-condition may define in which cases this function is to be applied. For example the following function definition states that *NewAircraft* is the function

[5] Visit `http://www.vdmtools.jp/` for more information on available VDM tools

[6] The examples used in this introduction are taken from the teaching books [FL09] and [FLM+05]. The interested reader is referred to these compendiums, to learn more details about *VDM-SL* and *VDM++*.

which takes an existing *RadarInfo* as well as a fresh *AircraftId* and *AircraftPoistion* and adds a mapping of this *AircraftId* to its position to the *RadarInfo*-set of mappings. The pre-condition states that this function may only be applied, if the *AircraftId* is fresh, i.e. it has not yet been recorded in *RadarInfo*:

```
NewAircraft: RadarInfo * AircraftId * AircraftPosition -> RadarInfo

NewAircraft(radar, airid, airpos) == radar munion {airid |-> airpos}

pre airid not in set dom radar
```

VDM-SL has a number of other features, such as implicit functions representation, the ability to model recursive structures and state-based modeling. However, a basic knowledge about data type and explicit function definition in addition to the object-oriented concept introduced in the next section should be enough to understand the models from Bryans et al. which are examined further on.

2.3.2 Object-Oriented VDM – VDM++

VDM++ is an extension of *VDM-SL* developed to accommodate object-oriented structuring and concurrency handling [FLM+05]. *VDM++* models are basically class definitions, which can be, according to the object-oriented paradigm, instantiated as an object with a unique identity. Building on the introductionary example from [FL09], a definition of the Aircraft-class might look like this:

```
class Aircraft

types

   AircraftModel = <BOENG767> | <BOING747_400> | <AIRBUS310>

   AircraftSerialNumber = seq of char
   inv snum == len sum > 0 and len snum <= 20;
   [...]

instance variables

   private model    :   AircraftModel;
   private serial   :   AircraftSerialNumber;
   private miles    :   nat;
   inv (model = <AIRBUSA310> or model = <AIRBUS320>) =>
       serial(17) = 'A';
   [...]

functions
   [...]
operations
   public Aircraft :AircraftModel * AircraftSerialNumber * nat ==>
   Aircraft

   Aircraft(mod,ser,mil) == (
      model := mod;
      serial := ser;
      miles := mil;
);
```

```
end Aircraft
```

The class definition starts with a definition of the types to be used in the model, in this case being a string collection type *AircraftModel* and a sequence of characters as *AircraftSerialNumber*. As shown in *VDM*-SL, invariants may constrain the structural properties of the defined types, for example stating that the character sequences of type *AircraftSerialNumber* have to be longer than zero and shorter than 20 characters. After the type definition the instance variables of the class are defined. An Aircraft object is supposed to instantiate the variables *model*, *serial* and *miles* with values of types *AircraftModel*, *AircraftSerialNumber* or *nat*, respectively. Again invariants may define consistency constraints that typically relate the variables such as in this example where the invariant over the instance variables states that all Airbus serial numbers have to start with an 'A'. Operations are defined similarly to its *VDM-SL* pendant, with the only difference that only an operation may update the instance variables in its own object or in an object referred to from within its own object. The example operation, which can be seen as a constructor for the class, updates the instances variables with the obtained input values.

With this basic understanding the models for dynamic coalitions from Bryans et al., explained in the next section 2.3.3 are understandable. In case of the occurrence of unexplained operators or syntax, footnotes will offer further explanation.

2.3.3 VDM and Dynamic Coalitions

In their technical report "Dimensions of Dynamic Coalitions" Bryans et al. propose a set of *VDM* models to capture various aspects of dynamic coalitions [BFJ+06]. They first identify a basic state and then define eight dimensions each dealing with certain aspects which could be of interest to modelers or software designers. The basic *VDM* state based on the first two dimensions, *Coalition membership* and *Information* shall be introduced here as this thesis takes a similar approach to the structure of the states of my proposed *ASM*-model. However, for simplification some model parts are modified or it is refrained from a exhaustive description of the model. The interested reader is referred to the works of Bryans et al. Also of interest is the dimension *Trust* which may later be utilized to enforce *trust-based access control* policies. Other Dimensions, such as *Authorisation Structure*, *Provenance* or *Time* could also be easily integrated into the *ASM*-model. However, they are not identified as critical dimensions in order to adress privacy and access control in dynamic coalitions and therefore refrain from a detailed description. Furthermore, in a recent publication Bab and the author of this thesis have proposed an extension of the *VDM* model of Bryans et al. which integrates basic *identity-based access control* with the model [BS11a]. This work will also be briefly summarized.

Basic State: Coalition Dimension

The basic state for coalitions with information (dimension *Information*) of Bryans et al.'s modeling approach consists solely of the following instance variables:

```
coals : map Cid to Aid-set := {|->};
agents: map Aid to Agent := {|->};
inv forall c in set dom coals &
        (coals(c) subset dom agents)
```

The basic state signature is based on *agents*, represented by agent identifiers *Aid* as well as coalitions as sets of agent identifiers, which reflects the situation of agents joining forces to form dynamic coalitions. A coalition is identified through

its *Cid*. Both ID types, as well as the Agent types, are at this point of the model seen as structureless tokes. The invariant (*inv*) states that only known agents may be part of coalitions. Invariants may be defined for variables and operations. In the following it will be refrained from explicitly defining all the invariants that usually are used to specify certain type properties.

Operations *Join()* and *Leave()* perform the obvious actions on coalitions.

```
Join : Aid * Cid ==> ()
Join(a,c) == (
   coals := coals ++ {c |-> coals(c) union {a}}
)

Leave : Aid * Cid ==> ()
Leave(a,c) == (
   coals := coals ++ {c |-> coals(c) \ {a}}
)
```

With these variables and operations the structure, as well the forming and dissolving of a dynamic coalition, may be expressed. However, as information sharing is often the main motivation of forming dynamic coalitions, modeling information items, as well as operations for sharing information, is vital. Thus, the next dimension is defined in chapter 3.

Information Dimension

In order to model agents sharing information it is necessary to define an *Information*-type as well as *Agent* and *Coalition* types which hold information items for themselves. Thus the basic type defintion is as follows:

```
public Agent       ::  info   : set of Information


public Coalition   ::  agents : set of Aid
                       info   : set of Information

public Information ::  item   : token;
```

The model abstracts from the actual content and structure of the information and considers information as unstructured data (*token-type*). The operations *Discover()* and *Share()* define an agents ability to create ("discove") information and share it with a coalition respectively. Note that the *mu*-operator is used to modify a single component of a record structure.

```
Discover : Aid * Information ==> ()
Discover(a,i) == (
   agents := agents ++ {a |->
   mu(agents(a), info |-> agents(a).info union {i})}
)

ShareInfo : Aid * Cid * Information ==> ()
ShareInfo(a,c,i)== (
   coals:= coals ++ {c |->
   mu(coals(c), info |-> coals(c).info union {i})};
)
```

With these operations a very basic behavior of dynamic coalitions considering its formation and the information sharing in the coalition may be modeled. However, for the goal of this thesis it was needed to formalize access control policies and the information sharing methods considering these. Therefore, the existing models from Bryans et al. had to be extended with the *Access Control Dimension*.

Trust Dimension

In the framework proposed in this thesis, *trust-based access control* is used to deal with the access control requirement in highly dynamic scenarios in which agents are not known prior to the collaboration, the infrastructure of the coalition may be too heterogeneous to enforce a traditional access control mechanism and the membership structure of the coalition changes too fast for any administrator to keep up with the adaptation of the access control policies. Trust has been recognized as a paradigm to tackle these issues (see section 1.2.1 and 2.2.4) and also Bryans et al. have integrated a basic notion of trust into their *VDM* models [BFJ+06]. As it is done in this thesis, they abstract away from the actual evaluation of trust, which is a deeply investigated research field itself with many a thinkable solution. Assuming that trust values have already been obtained, Bryans et al. model the *Trust Dimension* as follows:

```
coals : map Cid to Aid-set := {|->};
agents: map Aid to Agent := {|->};
inv forall c in set dom coals &
        (coals(c) subset dom agents)

public Agent       ::  aTrust   : map Aid to trustvalue
where trustvalue = real
```

The basic state remains the same, the Agent type however gets extended with the record field *aTrust* which for every agent provides a map from Agent-IDs to real numbers, representing a list of agents with their associated trust values, according to the consideration of the particular agent. Bryans et al. leave the usage of these trust values open for implementation, although they suggest that they might be used for definition of interaction requirements in a dynamic coalition e.g. stating that agents are expected to trust the other members in a coalition to some degree. In the framework proposed in this thesis it will be shown how these trust values may also be used to enforce access control in highly dynamic coalitions scenarios.

Access Control Dimension

Thus, in [BS11a] an extension of the *Information*-dimension from Bryans et al. was proposed, which integrates the main components for an *identity-based access control*. Since Bryans et al. already presented a *VDM*-model for specifying *XACML*-policies in other works it seemed straightforward to use this approach and integrate it with the dynamic coalition model. Therefore, the components and mechanisms proposed in this model are closely inspired by the *OASIS*-standard *XACML* and the corresponding *VDM*-model proposed by Bryans et al. in [BF07].

The main idea was that every time an agent wants to share information with a coalition, he will not only share the information item, but also the corresponding access control policies that restrict access to the information according to the owners definition (see fig. 2.11). In order to provide such a function both agents and coalitions had to be extended to hold a policy repository (it was called *PDP* for *policy decision point* in close resemblance to the *XACML* standard, although it

summarized the whole *XACML* component architecture in this record field and it was refrained from modeling PEP,PIP, etc.).

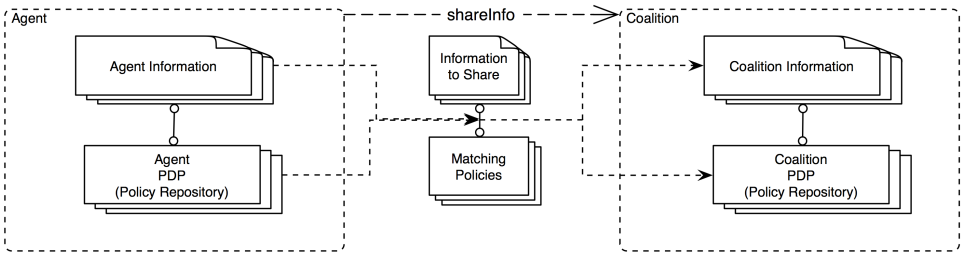

Figure 2.11: Access controlled information sharing model according to [BS11a].

The instance variables remained the same as in the *Information*-dimension. However, the extended types looked as follows:

```
public Agent       ::  info   : set of Information
                       aac    : PDP;

public Coalition   ::  agents : set of Aid
                       info   : set of Information
                       cac    : PDP;

public Information ::  item   : token;
```

The PDP of a coalition and the PDPs of the agents are related as follows: Upon every call of ShareInfo(), the related access control policies are together with the information to be shared, stored in the coalitions PDP. This way access control requirements of the agents may be enforced on coalition level. Furthermore, types and components for the *XACML*-policies are defined as follows:

```
public Rule    ::  target    : [Target]
                   effect    : Effect;

public Target  ::  subjects  : set of Aid
                   resources : set of Information
                   actions   : set of Action;

public Request ::  target    : Target;

public Action = (<WRITE> | <READ>);
public Effect = (<PERMIT> | <DENY> | <NOTAPPLICABLE>);
```

According to the OASIS-standard *XACML, policies* consist of *rules* which in turn consist of an optional *target* and an *effect*. Targets consist of *subjects*, in this case agents, which want to access certain *resources*, in this case information. *Actions* define the type of access in question, which was limited to *read* and *write* for sake of simplicity. When an access (*request*) matches a rule target, the *effect* is being returned which may be *permit*, *deny* or *not applicable* in case the target of the rule does not match the request. If a rule does not contain a target it will be evaluated for each request.

Policies are sets of rules including a combination algorithm which combines different effects in case that more than one rule matches a request. Here only the basic *XACML* combining algorithms *deny overrides* and *permit overrides* were

considered. Policies have obligatory *targets*, such that only matching requests are evaluated. The effect of a policy is returned to the *Policy Decision Point (PDP)* which combines the different effects to only one effect using the above-mentioned combining algorithm. Each request for access on information is evaluated through the PDP which ensures policy compliance.

```
public PDP    ::  policies      : set of Policy
                  policyCombAlg : CombAlg;

public Policy ::  target        : Target
                  rules         : set of Rule
                  ruleCombAlg   : CombAlg;

public CombAlg = (<DENYOVERRIDES> | <PERMITAOVERRIDES>);
```

2.4 Abstract State Machines (ASM)

> "Part of the essential difficulty of software development is that the simple ideas behind algorithms get obscured by encoding them in a language. If the original ideas, be they simple or difficult, can be expressed at the same level of complexity using *ASMs*, we have achieved a good thing." [HW02]

Abstract state machines started out as a means to formalize and simulate arbitrary algorithms in an abstract, meaning code-independent way. Gurevich sought an improvement of Alan Turing's thesis [Tur36], with what he first called *evolving algebras* and later entitled *Abstract State Machines*:

> "The original idea was to provide operational semantics for algorithms by elaborating upon what may be called the implicit Turing's thesis: every algorithm is simulated by an appropriate Turing machine [...] Turing machines give operational semantics to algorithms. Unfortunately this semantics is not very good. Turing machine simulation may be very clumsy. In practice, one step of the algorithm may require a long sequence of steps of the simulating Turing machine. I was looking for machines able to simulate algorithms much more closely. In particular, the simulation should be *lock-step* so that the simulating machine makes only a bounded number of steps to simulate one step of the given algorithm. Evolving algebras [...] are supposed to be such machines." [Gur93]

Thus, the formalism of abstract state machines is a method for the modeling of arbitrary algorithms, without having to deal with language-dependent concepts which may distract from the abstract concept of the algorithm itself. The main idea is to equip algebraic structures, such as structures in first-order logic defined by Tarski [JT52], with a logic which allows for the representation of transitions in between algebras of the same signature. By means of updates of the functions and relations that an algebra contains a definition of programs or algorithms becomes possible. Those updates may proceed in a sequential, parallel or distributed manner, defining sequential, parallel or distributed abstract state machines respectively.

The deduced *ASM* thesis states, in allusion to the implicit Turing's thesis that:

> " [..] every algorithm is behaviorally equivalent to an abstract state machine and in particular is simulated step for step by that machine." [BG03]

Gurevich gave axiomatizations of sequential, parallel and interactive (as a generalization of distributed algorithms) which reflect the common understandings and those concepts and proved the *ASM* thesis for those [BG03, Gur00, BGRR07]. *ASMs* have since been applied and experimented with in various theoretical and practical context and proven to be an effective tool to model, specify and document various computing systems, for example programming languages [GH93], architectures and to validate language implementations [BG94].

Since the level of abstraction for formulating the algebras is arbitrary, such algorithms may include implementable (that means computer based) algorithms in the same way as for example algorithms describing a certain course of action or an algorithmic procedure of the real world, which does not necessarily have to be implementable [BS03, GKOT00]. Exactly that flexible level of abstraction has made *ASM* applicable in a number of fields outside of software and information science, such as sociology [BGK+05] or mathematics [BS03].

2.4.1 Why ASMs for DCs?

In order to define and formalize the concept of privacy-aware dynamic coalitions a formalism was needed which provides freedom of abstraction to model structural details on an algebraic level such as in [BFJ+06] (see section 2.3.3) but also allows for specification of agent behavior as well as the interaction processes in between them. Other formalisms which provided these features exist with the most prominent one being *Algebraic High Level (AHL) Nets*, which is an extension to regular *Petri-Nets* integrating algebraic structures into the state description. However, in *ASMs* allow for a more detailed definition of transition conditions as well as the freedom to dynamically extend the algebraic structures during runtime. The latter is identified as an essential aspect for the the definition of dynamic coalitions, since their dynamic aspects produce unexpected behavior such as the extension of universes or the change of function values, which is not possible to formalize in other formalisms such as *AHL-nets*.

The specification of an *ASM* takes place in a intuitive way, defining agent programs, which may then interact with each other by means of global variables. In this sense *ASM* operates like regular concurrent programming languages and is therefore easily conceivable for any software engineer or computer scientist. Nevertheless, *ASM* is not to be seen as yet another programming or simulation language. It is rather to be regarded as an abstraction of algorithmic concepts, resulting in mathematical, clear and unique definitions of algorithms or processes. In this sense it is language independent and at the same time powerful enough to model any arbitrary algorithm [Gur93, BG03, Gur00, BGRR07]. This independence, makes way for an understanding of the algorithmic processes depicted in an ASM, without the need for computer science knowledge, while at the same time being formal enough to support software engineers with the modeling of software architectures. Thus, *ASM* may be utilized not only as a modeling tool but also as a means to bridge the gap of understanding between software engineers and domain experts (e.g. medical doctors).

Because of the above mentioned concepts, the *abstract state machine* formalism is seen as the ideal to provide a framework for dynamic coalitions which formally defines the concept itself and thereby makes validation and verification possible. Furthermore, according to Börger and Stärk, *ASMs* may be utilized by system engineers to gain a:

> "[...] correct and complete human-centric task-formulation, called the *ground model*, which [...] is easy to understand, and constitutes a

> binding contract between the application domain expert (in short: the customer) and the system designer." [BS03]

Since bridging the gap between system engineering and domain experts such as between software developers and medical personal is a main part of the motivation of this work, the concept of the *ASM ground model* is essential: In this respect the proposed framework may be regarded as a meta-model for *ground model* construction, defining in an abstract, application-independent manner which components a *ground model* for a privacy-aware dynamic coalition scenario should have and which interaction processes should be defined.

Thus, the model proposed in this work makes use of the *Abstract State Machines (ASM)* methodology and is illustrated and implemented in the *CoreASM*-language. In the following sections an introduction to *ASMs* is given. For this introduction a mathematical syntax will be utilized as it has been done in various *ASM* definitions [BS03, GKOT00]. However, in the core parts of this work, the proposed models rely on the syntax which is used by the *ASM*-tool of choice: *CoreASM*.

Similar to the chronological history of *ASMs*, the introduction will first start out with a definition of a *basic ASM* for a monolithic *ASM* program, with possibilities of simultaneous parallel actions with or without determinism. This basic definition was later extended to the definition of a *distributed ASM* (also called *multi-agent ASM*) for formalizing asynchronous interaction of multiple agents.

2.4.2 Definition: Basic ASMs

Here definitions taken from [BS03, GKOT00] and [Far09] are summarized in an incremental manner, in order to gain a common understanding of basic *ASMs*. Other definitions may vary slightly to insignificantly in syntax but follow a similar structure. This particular representation was chosen here because it also serves as the foundation of the *CoreASM*-tool, first proposed by Farahbod in [Far09].

Signature

A signature Σ, is a finite set of function names with an arity. Functions with arity zero, are called *nullary* functions or *constants*. Functions with arity bigger than zero, can be seen as functions with a number of parameters in accordance with the function arity. The nullary functions *true*, *false* and *undef* are defined for every signature Σ. Functions may be defined as *static* or *dynamic*, with the latter meaning that the values of the function may change from state to state. States are defined as follows.

States

A state $\mathfrak{A}$ of a signature Σ is a non-empty set X (the union of all universes defined in $\mathfrak{A}$, the so called *superuniverse* of $\mathfrak{A}$) with an interpretation $f^{\mathfrak{A}}$ for all functions f in Σ, with the following constraints: If f is an n-ary function name, then $f^{\mathfrak{A}} : X^n \to X$ and if c is nullary, i.e. c a constant in Σ, then $c^{\mathfrak{A}} \in X$. $\mathfrak{I}$ is a set of initial states.

Location

A location l in a state Σ is a pair $(f, < a_1, ..., a_n >)$ with a n-ary function name f and values $a_1, ..., a_n$ of the *superuniverse* X.

Content of a Location

The result of interpretation of a function f with values $a_1, ..., a_n$, i.e. $f^{\mathfrak{A}}(a_1, ..., a_n)$ is the content of a location.

Update of a Location

An update of the form (l, v) means a change of the content of location l to the value v which is a value of the *superuniverse* X.

Consistent and Inconsistent Update Sets

An update set is a set of updates. An update set is called *consistent*, if it has no clashing updates: for any location l and all elements v, w, it is true that if $(l, v) \in U$ and $(l, w) \in U$, then $v = w$. An update set for which this is false, is called *inconsistent*.

Rule Declarations

A rule declaration is an expression of form: $R(x_1, ..., x_n) = P$, with R being the name of the new rule, P a transition rule (see section 2.4.3) and $x_1, ..., x_n$ containing all free variables in P.

Basic *ASM*

A basic *ASM* M is a tuple of form $(\Sigma, \mathfrak{I}, R, P_M)$ with elements as defined above, $\mathfrak{I}$ being a set of initial states and P_M standing for a distinguished rule with no free variables, also called the Program of machine M.

2.4.3 Definition: Transition Rules

The program P_M of an *ASM* M is defined through means of *ASM* transition rules, such as:

Update Rule

A rule of form: $f(a_1, ...a_n) := t$ which results in a set of updates $\{(f(a_1, ..., a_n), t^{\mathfrak{A}})\}$ with $\mathfrak{A}$ being the current state of the machine and $t^{\mathfrak{A}}$ the evaluation of t in $\mathfrak{A}$.

Block Rule

A rule of form: P **par** Q which evaluates the results of rule P and rule Q in a parallel manner and returns the union of both computed sets of updates.

Conditional Rule

A rule of form: **if** Φ **then** P **else** Q with Φ as a *boolean* term, which if *true* is followed by evaluation of P and else wise by evaluation of Q.

Let Rule

A rule of form: **let** $x = t$ **in** P which assigns the value t to the variable x[7] and then evaluates P.

Forall Rule

A rule of form: **forall** x **with** Φ **do** P which for every x that satisfies Φ executes P in parallel. The result is an update set of all the update sets produced by the parallel execution of P over the different values of x.

[7]The variables in **let**, **forall** and **choose** are logical read-only variables, meaning that their values are not subject to updates of transition rules. They are not stored in the state but in a finite environment.

Choose Rule

A rule of form: **choose** x **with** Φ **do** P **ifnone** Q which non-deterministically choose an x that satisfies Φ and executes P, if such an x exists, else wise Q is executed.

Sequence Rule

A rule of form: P **seq** Q which first applies the update set produced by P. In this state with applied updates of P, then Q is executed. Note that Q is only applied *if and only if* the produced update set from P is consistent. If the update set U_P of P is inconsistent the produced update set of P **seq** Q is the update set U_P only.

Call Rule

A rule of form: $R(a_1, ..., a_n)$ which calls the rule R with the parameters $a_1, ..., a_n$

2.4.4 Definition: Distributed ASMs

A distributed *ASM* (*DASM*) M^D is defined as a set of *ASMs* each of them being the program of a set of computational agents, where the executions of the machines are asynchronous. The agent-set may change during the run of an *ASM* and is therefore called dynamic. Agents in a *DASM* may interact with each other and with the environment through means of accessing and changing a global machine state. Computation steps of a single agent are called *moves*. If moves of computational agent do not conflict with each other, they may be executed jointly during one computational step of M^D. Thus, conflicting moves must be ordered so that the do not appear in the same step of M^D. The notion or the *partially ordered run* provides this ordering. Consider [BS03] for a detailed definition of a *partially ordered run*.

2.4.5 Other ASMs

Various variants of *ASMs* have been proposed in literature, such as *Control State ASMs*, representing a normal form of synchronous UML activity diagrams or *TurboASM* which provide rule declaration as composition of submachines as well as parameterized submachines and iteration and recursion of rules. Especially the latter will be used extensively in the models of the framework at hand. See [BS03] for detailed information on the various available *ASM* definitions.

2.5 CoreASM

CoreASM is an open source project brought to live by Farabod et al. in [FGG07]. It offers an executable and extensible *ASM* language, which allows for an actual programming with *ASM*. Both, its openness and its extensibility makes it a fitting tool for the purposes of this work as the extensibility or generalization of the proposed framework through further research is highly anticipated.

2.5.1 CoreASM Architecture

In order to provide extensibility, which allows for more freedom in designing *ASM* specification, *CoreASM* has been designed as a plugin-based architecture, with a minimal set of plugins at its core (see figure 2.12), providing the very basic functionality an *ASM* tool has to provide and some optional plug-ins as well as open interfaces to include hand-made plugins that capture relevant functionality according

to one's needs. The *CoreASM*-engine operates on these plugins may communicates with external applications such us a graphical code editor or other custom GUIs.

Currently the two main *CoreASM*-engine drivers are the *CoreASM* Eclipse Plug-in for the Eclipse development environment and the command-line *CoreASM* engine driver *carma*. As the former provides syntax highlighting, the following sections will stick to these syntax highlighting conventions in order to improve readability.

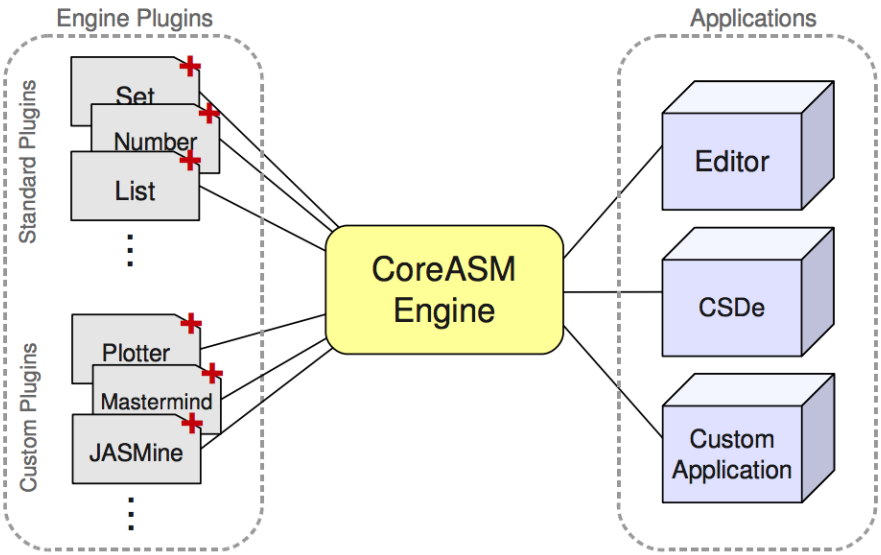

Figure 2.12: *CoreASM* extensible architecture diagram taken from the *CoreASM* language user manual.

2.5.2 CoreASM Specification Structure

In the following an overview of the structure of *CoreASM* specifications will be presented (see figure 2.13) including the basic and optional plugins which are to a great extent used in the models of this thesis offers as well as operators and rule forms to be used to specify basic as well as distributed *ASMs*. The definitions are taken from the *CoreASM language user manual*[8].

Every *CoreASM*-specification starts out with the keyword "CoreASM" assigned to the name of the specification, which is followed by a call of the required plugins using the *use*-keyword.

Loading Plugins

Expressions and rules, as well as extensions to interact with environment, with the user or with external applications, are implemented through means of plugins. While every plugin may be loaded individually, two compilation packages contain set of plugins. The *BasicASMPlugins* plugin set in *CoreASM*, implements rule forms and expressions to provide basic *ASM* functionality according to the rules mentioned above. *StandardPlugins* contains the set of *BasicASMPlugins* and adds expressions that are frequently used in *ASM* specification but are not regarded as basic *ASM* expressions, such as: *extend*-rule to import fresh elements to universes,

[8]See `http://www.coreasm.org/downloads/CoreASM-UserManual-DRAFT.pdf` for the *CoreASM* language user manual

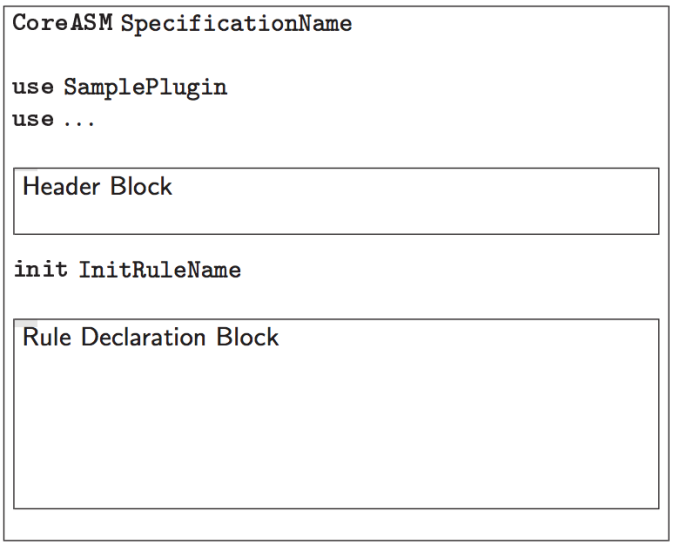

Figure 2.13: *CoreASM* specification structure pattern taken from the *CoreASM* language user manual.

turboASM-rules to allow parameter passing to and from rules, *input* and *output*-rules and more. *BasicASMPlugins* and *StandardPlugins* are loaded through means of

```
use BasicASMPlugins
```

or

```
use StandardPlugins
```

respectively. Additional plugins are provided by *CoreASM* or may be implemented by hand and may then be called in a similar fashion

Header Block

Definition of structural properties of the *ASM* takes place in the header block. Which elements are defined is dependent on the plugins used. According to notational custom in *ASM* literature, *CoreASM* specifications do not necessarily have to explicitly define the used structures. Thus, the *CoreASM*-kernel does not define anything for the header block. Implicit definition means that upon execution of an *ASM*-program, the *CoreASM* engine internally creates universes, function names and values, according to their use in the program specification. This may be very comfortable for quick analysis or specifications of algorithms. However, for the purposes of this thesis the explicit signature definition is more practical as not only algorithmic properties are to be defined explicitly but also the structure of the coalition itself. In addition to that a certain amount of type checking is made possible through the definition of universes, functions and values. Therefore the *Signature-Plugin* will be itself, itself being part of the *StandardPlugins*-set, which is introduced in section 2.5.5.

Init-Rule

After the header block the specification is initiated through the call of a special rule that serves as initiation rule. The rule itself, along with all the other rules that constitute the *ASM*-program is defined in the rule declaration block.

Rule Declaration Block

In the rule declaration block the *ASM* rules are defined. The rule forms are defined by the kernel and by the optional plugins. The definition enlistment here is restricted to the particular plugins that are used in the framework for privacy-sensitive dynamic coalitions. For a detailed definition of all available plugins see [Far09].

2.5.3 CoreASM Kernel Rule Forms

The kernel of *CoreASM* defines amongst others the following rules forms:

Location Update

According to the **Update Rule** in section 2.4.3 a rule of the form

loc := *value*

assigns the value of *value* to the location *loc*.

Skip Rule

The skip rule is executed by the keyword

skip

and is a rule form that does nothing. It may be seen as an empty rule or NoOp.

2.5.4 BasicASMPlugins Rule Forms

The plugin package *BasicASMPlugins* provides the following rule forms:

Block Rule

According to the **Block Rule** defined in 2.4.3 a rule of form

par $rule_1$ $\underbrace{rule_2 \ \ldots \ rule_n}_{optional}$ **endpar**

evaluates the given rules in parallel. The update set generated by this is the union of all the updates generated by the given rules.

Conditional Rule

According to the **Conditional Rule** defined in section 2.4.3 a rule of form

if *value* **then** $rule_1$ $\underbrace{\textbf{else}\ rule_2}_{optional}$

evaluates $rule_1$ only if *value* is **true** and optionally evaluates $rule_2$ elsewise. The rule expects *value* to be a boolean value.

Choose Rule

According to the **Choose Rule** defined in section 2.4.3 a rule of form

choose *id* **in** *value* $\underbrace{\textbf{with}\ guard}_{optional}$ **do** $rule_1$ $\underbrace{\textbf{ifnone}\ rule_2}_{optional}$ $\underbrace{\textbf{endchoose}}_{optional}$

chooses an element from *value* which satisfies *guard*, assigns it to *id* and evaluates $rule_1$. *value* is expected to be an enumerable element, i.e. a collection of other values, such as sets, universes, etc. The optional **ifnone** clause evaluates $rule_2$ if no element that satisfies *guard* can be found.

Forall Rule

According to the **Forall Rule** defined in section 2.4.3 a rule of form

forall *id* **in** *value* **with** *guard* **do** *rule* **endforall**
optional *optional*

assigns the element to *id* for each element of *value* which satisfies *guard* and evaluates *rule*. *value* is expected to be an enumerable element, i.e. a collection of other values such as sets, universes, etc.

Predicate Logic

The Predicate Logic plug-in provides the expressions $\forall$ and $\exists$ in the following form:

forall *id* **in** *value* **holds** *guard*

and

exists *id* **in** *value* **with** *guard*

are boolean expressions which hold if *guard* holds for all the elements of *value* or there exists at least one element in *value* that satisfies *guard* respectively. *value* is expected to be an enumerable element, i.e. a collection of other values such as sets, universes, etc.

$rule_1$!= $rule_2$

provides a not equal expression which is defined on all elements and may also be defined by using the **not**-expression:

not (*rule*$_1$ = *rule*$_2$)

Furthermore, the following expression

$value_1$ bin-op $value_2$

performs a binary operation on the given values, with operators `or`, `xor`, `implies`, `memberof`, `notmemberof`, where the latter two expect $value_2$ to be enumerable.

2.5.5 StandardPlugins Rule Forms

The plugin package *StandardPlugins* amongst others provides the following rule forms and expressions:

Extend Rule

A Rule of form

extend *value* **with** *id* **do** *rule*

imports a new element if *value* is a universe, assigns it to *id* and evaluates rule. If *value* is a background (such as *String*, *Number*, etc.), it gets the default element from the background, assigns it to *id* and evaluates *rule*. The resulting update set for the latter case is the union of the updates generated by *rule*. In the former case a single update is added, which adds the imported element to the universe *value*.

TurboASM Rules

TurboASMPlugin adds various rule forms to provide sequential composition of *ASM*, iteration and recursion as well as parametrized submachines.

According to the **Sequence Rule** defined in section 2.4.3 a rule of form

seq *rule*$_1$ **next** *rule*$_2$
optional

applies the generated updates of *rule*$_1$ in a virtual state, in which then *rule*$_2$ is evaluated. The resulting update set is a sequential composition of the two generated update sets. In a similar way the following rule executes a greater number of rules in a sequence.

seqblock *rule*$_1$ *rule*$_2$ *...* *rule*$_n$ **endseqblock**
optional

Rule recursion may be implemented with the following rule

iterate *rule*

which repeatedly evaluates *rule* until the produced update set is either empty or inconsistent. The resulting update set is the accumulation of all the updates until that point. If the last update set is inconsistent, the resulting update set of this rule may be inconsistent. Recursion over a condition may be written as

while (*value*) *rule*

which repeatedly evaluates *rule* as long *value* is *true* and may therefore also be written as:

```
iterate
    if value then rule
```

Returning of a result of a rule execution may be achieved through the following two expressions:

result := *value*

is written in the rule which is supposed to return *value*. The return *value* of a rule may be assigned to a location through:

loc <- *rule*.

In a similar manner this rule

return *value* **in** *rule*

provides a return *value* which has to be defined in *rule*. The only difference is that all the updates of *rule* are applied in a virtual state, which is used to evaluate *value*. After this evaluation however, the virtual state, as well as the updates, are discarded, i.e. this expression cannot change the global state of a machine because all updates are “forgotten” after evaluation.

Input and Output

The *IOPlugin* provides rules and expressions for basic string input and output.

print *value*

prints out *value* to the environment, i.e. the standard output of the engine driver (for example the *CoreASM* dialog window in the *CoreASM Eclipse Plugin*, or the terminal window in the *Carma* command line editor for *CoreASM*).

Input strings from an environment (for example a user interaction with the *ASM*-model) may be recorded through the function *input* : STRING → STRING, which outputs a parameter *String* and then waits for a *String* input which may then be assigned to a location:

loc := **input**(*stringvalue*)

Set Background

The *SetPlugin* provides a background SET, with a number of functions and expression forms.

$\{\underbrace{\texttt{value}_1\texttt{,...,value}_n}_{optional}\}$

creates a set that includes the listed values, which are required to be basic terms or to be surrounded in parentheses.

In order to allow for parallel incremental updates of sets the *SetPlugin* provides two rule forms:

add *value* **to** *loc*

and

remove *value* **from** *loc*

with the obvious semantics, whereas *loc* is required to be a location in the state with its value being a set.

Signature Plugin

As mentioned in section 2.5.2, the *SignaturePlugin* allows for explicit definition of state structures, providing the following expressions for defining universes, finite value sets and function over the former:

universe id $\underbrace{\texttt{= }\{\texttt{id}_1\texttt{,...,id}_n\}}_{optional}$

defining a universe, i.e. an extensible set with optional elements.

enum id = $\{\texttt{id}_1\underbrace{\texttt{,...,id}_n\}}_{optional}$

defining an enumerated, finite, i.e. inextensible set with at least one value.

function $\texttt{id}_f$ = $\underbrace{\texttt{id}_{u1}\texttt{ *...* id}_{un}}_{optional}$ -> $\texttt{id}_r$

defining functions as mappings from optional parameters to an output value. Functions without input parameters are constants which hold values of the defined background

2.5.6 Other Plugins

BasicASMPlugins defines further plugins, such as *LetRulePlugin* which provides a **let** ..**in**-rule or *NumberPlugin* which sets up a number background, with basic operators and expressions for handling natural, integer and real numbers. Backgrounds for strings, collections, lists, queues, stacks and maps may be defined through plugins of the *StandardPlugins*-package. Additional plugins allow for integration of a time notion or for the definition of mathematical formulas. However, all those plugins are not extensively used in the proposed framework of this thesis. Thus it will be refrained from a detailed definition and the interested reader is referred to [Far09].

2.5.7 Validation of CoreASM Specifications through Simulation

The proposed framework of this thesis aims at providing a modeling tool for dynamic coalitions that helps to understand and define the key components of a *ground model* for a dynamic coalition in a formal manner. In the spirit of *model driven development* this *ground model* may then be refined to close the gap between the abstract formal model and the software implementation. Along those refinement steps it is essential to have the possibility to validate certain properties of the system, such as while the model becomes more concrete the key features of the *ground model* are not lost. For example, in privacy-sensitive dynamic coalition scenarios the validation of the access control policies is extremely important in the conceptual design phase as well as in the implementation phase of the software development.

The proposed framework serves as a platform for the realization of these validation requirements. *CoreASM* provides an execution engine, which makes abstract *ASM* specification executable. Therefore it is a helpful tool for experimental validation through simulation. The evolution of the computing steps of the *ASM* specification may be visualized through the *output*-plugin provided by *CoreASM*, which basically allows for command line outputs. Furthermore, the *observer*-plugin provides an XML representation of all the states produced by an *ASM* run and may therefore be exploited for graphical representation of *ASM* runs. Through a GUI one could explore the various states created during an *ASM* run and investigate the evolution of universes and functions along the way. In this work a first prototype for this kind of *state exploration*-GUI is provided.

Note that all the validation efforts of this thesis are based on simulation of the proposed models. While, as mentioned above, the formal nature of the framework obviously allows for the integration of sophisticated testing methods, the application of the latter was not in the scope of this thesis as realizing this task may be subject of a thesis of its own. However, the testing of the proposed models is envisioned in future research and will be investigated in the last chapter of this thesis.

Chapter 3

A Formal Framework for Privacy-Sensitive Dynamic Coalitions

In this chapter the core of this thesis is presented: **a formal modeling framework for privacy-sensitive dynamic coalitions**. In particular the integration of the various access control mechanisms into the concept of the dynamic coalitions will be examined. The work is inspired by previous research of Bryans et al. who proposed a set of formal models for structural properties of dynamic coalitions, each according to a certain perspective or "dimension" of interest, such as coalition membership, information transfer, etc.[1] However, the models of Bryans et al. solely address structural aspects of dynamic coalitions in the formal specification language *VDM*. Exceeding these structural features the consideration of coalition processes is of high significance in dynamic coalitions. Therefore, the scientific challenge of this thesis lies in the integration of structural properties as well as dynamic aspects of dynamic coalitions into one single formalism. This integration has to be defined for dynamic coalitions in their basic form, as well as for privacy-sensitive dynamic coalitions, for which common access control mechanisms have to be abstractly defined in the same formalism. To this end, this work utilizes *Abstract State Machines (ASM)* to enrich the models of Bryans et al. with formal process aspects. Bryans' *VDM*-model can be seen as the initial inspiration for the underlying states of the *ASM* and processes may be represented as state transitions in the *ASM*. However, in order to integrate access control into the models the structure itself had to be modified also. Thus, speaking in terms of Bryans et al. the *dimension access control* was created.

The proposed framework consists of various models: A basic dynamic coalition model, which formally defines the key components and operations that are characteristic of a dynamic coalition. This formal model is regarded as the basic definition of what dynamic coalitions are and therefore constitutes the foundation of all other models following in this thesis. It will later be extended and modified in order to present models to integrate the various access control mechanisms: *identity-based access control (*IBAC*)*, *role-based access control* (*RBAC*), *attribute-based access control* (*ABAC*) and *trust-based access control* (*TBAC*).

The following sections are organized as follows: In the first section the *ASM* structure or signature for the basic dynamic coalition model is introduced, along with the universes, function and operations. [2] The signature definitions in the

[1] See section 2.3.3 for an introduction to the models of Bryans et al.

[2] In order to distinguish *ASM* rules from later defined access control rules, this thesis will refer *operations* when meaning the possible actions and interactions that may take place in a dynamic

following sections do not necessarily have to be provided in order to model basic dynamic coalition scenarios in *CoreASM* because *CoreASM* infers the universes and functions needed for the transition rules automatically. However, this model explicitly presents the signature in order to increase the understanding of the *ASM*-rules in later sections.

Then the integration of access control mechanisms into the *ASM*-model of dynamic coalitions will be presented, thereby identifying the *dimension access control*. Those mechanisms will be integrated by adding or modifying universes, functions and operations according to the requirements at hand, resulting in a new, fully operational *ASM* model for each access control mechanism at a time. In the last section of this chapter it will be demonstrated how simulations may be conducted on these models, utilizing *CoreASM* and a handmade tool.

3.1 Basic Dynamic Coalition Model

In order to model a basic dynamic coalition, without considering its access control components, the following main model requirements have to be identified:

Model Requirements

Structural requirements:

- Need to represent agents in a dynamic manner, i.e. allowing for the arrival of new, previously unknown agents during runtime.
- Need to represent structureless information items which agents carry and may create during runtime.
- Need to represent coalitions which are to be seen as sets of the agents that are to collaborate. Coalitions are to be agents too, thereby making it possible to model coalitions that interact with other agents or coalitions and also allow for nested coalitions.

Operational requirements:

- Need for methods for the creation of new agents and new information items that agents carry as well as deletion of information items from an agent's repository. Only information owners may delete information items from their own or another agent's information repository.
- Need for methods for the adding and removing of agents to a coalition.
- Need for a method for the sharing of an information item with a coalition, thereby making it accessible to access requests from other agents in the coalition.
- Need for a method which provides an answer to an access request for an information item of a coalition with the only limitation that solely agents inside a coalition may access the contained information. This rule will later be extended with the corresponding access control mechanisms and components.

The presentation of the model is organized as follows: First the universes and functions needed to formalize the structure of a basic dynamic coalitions are depicted, followed by an enlistment of the operations which provide the functionality on these universes. The concluding model will serve as a basis for the extension and modification for later considerations of access control. The following structure corresponds with Bryans et al.'s *Information Dimension* [BFJ+06] as the later focus of my work lies on information sharing in dynamic coalitions. However, Bryans et al. did not consider operations such as those for sharing or requesting operations. Therefore, the following model may already be seen as an extension of existing works.

coalition model

3.1.1 Basic Dynamic Coalition - Signature

The basic signature of an abstract state of dynamic coalitions consists of:

```
universe Agents
universe Information
```

The first line describes the extendable set (**universe**) of **Agents**. Note that it is written in a bold font, indicating that it is a predefined universe of *CoreASM*, specifically representing the agents of a distributed *ASM*. However, as mentioned above this work chooses to explicitly introduce this universe in the model, even if it is redundant. The ability to extend **Agents** through import of fresh elements from an infinite reserve set at runtime is characteristic of a dynamic coalitions, where agents may come and go at any time, i.e. during runtime.

The enumerable sort *Effect* will store the possible answer to access requests:

```
enum Effect = {permit, deny}
```

The set of all information items in the specification is denoted by the universe *Information*. Agents may carry information, illustrated by the following mapping function:

```
function AgentInfo: Agents -> SET /*of Information*/
```

SET is a basic *CoreASM*-type and stands for an unordered untyped set of elements. The fact that in this particular case only sets of *Information* items are meant can be enforced through additional invariants or through the transition rules which will be defined later on. Nevertheless, in order to increase the understandability of the model comments were added here which describe the types of the set elements envisaged for the sets in question.

A coalition is a special agent for which the following partial function is not undefined:

```
function CoalitionMembers: Agents -> SET /*of Agent*/
```

meaning that as soon as the function *CoalitionMembers()* maps from an agent to a set of other agents this particular agent is considered a coalition. This perspective of a coalition being an agent itself is important if the model is to allow interactions in between coalitions or allow for coalitions to be able to contain sub-coalitions as members.

A further function has to be introduced: The *InfoOwner()*-function is set to the creating agent and is later needed in order to ensure that an agent always has access to his own information. Also the *SharedWith()* function records a set of all agents, with which an information item has been shared, which will later be needed for the update of policies concerning information items that have already been shared.

```
function InfoOwner: Information -> Agents
function SharedWith: Information -> SET /*of Agents*/
```

The auxiliary function *CallByName()* is a helper function which maps a string name passed as a parameter to an element, for example the name of an agent to an agent universe element. The definition simplifies later calling of the elements by their defined name.

```
function CallByName: STRING -> ELEMENT
```

In an analogue way the following function returns the name of an element. It serves as a helper function in order to improve dialog outputs in *CoreASM*, making it possible to output the name of an agent, an information etc.

```
function Name : ELEMENT -> STRING
```

3.1.2 Basic Dynamic Coalition - Operations

ASM rules represent the possible operational behavior, i.e. the possible transitions of a basic dynamic coalition. *TurboASM* rules may receive parameters for the execution of their program.

A method for creating new agents is necessary in order to introduce new agents to a dynamic coalition during its runtime. The following method creates an empty agent where "empty" stands for the information set that a newly created agent carries initially. Note that no explicit method for the creating of coalitions is defined, due to the fact that in this model coalitions are seen as agents for which the *CoalitionMembers()*-function is not empty. Therefore, the initial set of this function for an agent is always empty and may later be filled with other agents, which is modeled further down:

```
rule  SetUpEmptyAgent(agentname) = {
  extend Agents with a do {
    CallByName(agentname) := a
    Name(a) := agentname
    AgentInfo(a) := {}
    CoalitionMembers(a):={}
  }
}
```

In this model agents are considered to be the originators of information items. Therefore a call of the *CreateInfo()*-rule will have the calling agent as the first argument. This rule creates an information item and assigns it to the agents information set through an update of the *AgentInfo()* function of the agent. It also defines the owner of the information item as well as the set of agents with whom this information has been shared with, initially only the owner himself.

```
rule  CreateInfo(ag,infoname) = {
  extend Information with i do {
    CallByName(infoname) := i
    Name(i) := infoname
    InfoOwner(i):=agent
    SharedWith(i):={agent}
    add i to AgentInfo(agent)
  }
}
```

Information items may be removed from an agent's repository. However, only the information owner is allowed to remove his information, either from his own repository or from a coalitions repository that he shared his information with. [3]

```
rule DeleteInfo(ag1, ag2, i)= {
  if InfoOwner(i)=ag1 then
    remove i from AgentInfo(ag2)
}
```

[3] One could consider methods for deleting agents, information or more from their corresponding universes in order to save memory space. This approach would be similar to some kind of garbage collector in a normal programming language such as *Java*. However, this is rather a technical consideration and does not lie in the scope of this model. Since the definition of such methods is straightforward and the models for the time being are not extensively memory consuming it was chosen to omit this kind of operation.

The model at hand supposes that the creator of an information item is its owner and that there might be only one creator for each item. However, the owner field may easily be modeled as a set of agents, allowing for multiple owners. For reasons of simplicity this model chooses the first approach. Nevertheless, it might be useful to change the owner rights of one's information in order to allow someone else to administrate the information item. The following allows an owner to change the owning rights:

```
rule ChangeInfoOwner(ag1,ag2,i) = {
  if InfoOwner(i)=ag1 then
   InfoOwner(i):=ag2
}
```

Agents may join coalitions. This happens through state updates of the *CoalitionMembers()*-function at the according locations. Consider the following *ASM* transition rule as a method for adding an agent to a coalition respectively deleting an agent from a coalition:

```
rule AddCoalMember(ag, co) = {
  add ag to CoalitionMembers(co)
}

rule RemoveCoalMember(ag, co) = {
  remove ag from CoalitionMembers(co)
}
```

An agent may share his information with a coalition if and only if he belongs to that coalition.[4]

```
rule ShareInfo(ag,co,i) = {
  if InfoOwner(info) = ag and
  if exists agent in CoalitionMembers(co) with agent= ag then {
    add i to AgentInfo(co)
    add co to SharedWith(i)
  }
}
```

A *RequestInfo()*-rule serves as the basic method in order to determine if an agent is granted access to an information item or not. In the basic model, the access rights to an information item is solely determined through the membership in the coalition in which the information resides. The only exception to that rule is the info owner, which will always have access to his own information. In later sections this method alongside with others will be extended to meet the policy requirements of the corresponding access control mechanism. Furthermore, an access request may only be evaluated if the requested resource exists. If it does not, the rule will return a *notapplicable*-answer.

[4]The sharing of information is considered to be an action between an agent and a coalition: Only an agent that belongs to a coalition may share his information with the latter. This design decision has been made due to the intrinsic motivation of a coalition, which is to share information or other resources in order to achieve a common goal. Of course in reality agents also share information between each other without belonging to an enclosing group. However, this kind of sharing is regarded to already identify the emergence of a coalition, in this case between two single agents. Therefore, if two agents want to share information solely between them in this model they formally need to first establish a coalition. Since the focus of this model is to identify the processes of information sharing in a dynamic coalition this perspective becomes imperative.

```
rule RequestInfo(ag,co,res) = {
  if InfoOwner(res)=ag then result:=permit else
    if exists agent in CoalitionMembers(co)
    with agent= ag then
      if exists r in AgentInfo(co) with r=res then
        seq print Name(co)+": ACCESS GRANTED FOR "+Name(ag)+"!"
        next result:= permit
      else
        seq print Name(res)+ ": ACCESS NOT APPLICABLE! "+Name(res)+
           " NOT IN COALITION REPOSITORY!"
        next result:= deny
    else
      seq print Name(co)+": ACCESS DENIED! "+Name(ag)+" NOT
         IN COALITION!"
      next result := notapplicable
}
```

With these universes, functions and rules a basic dynamic coalition of agents who want to share information with each may be modeled. This model serves as the basic definition of a dynamic coalition and its characterizing interactions. The following sections will go further and consider access control mechanisms in order to enforce privacy in dynamic coalitions.

3.2 Dimension Access Control

The integration of structural and dynamic properties of dynamic coalitions together with their access control policies into one formalism will be outlined in this section. With the resulting models it will be possible to properly formalize privacy requirements for dynamic coalitions and make them verifiable. According to Bryan's et al. terminology of "Dimensions of Dynamic Coalitions" [BFJ+06] another missing dimension is investigated: The Dimension Access Control. To that end the aforementioned basic dynamic coalition model of section 3.1 will be extended with new universes and functions. Furthermore new rules are created and existing rules such as *ShareInfo()* or *RequestInfo()* will have to be modified to fulfill the access control requirements. Universes, functions and rules that will not be altered from the basic model will not be repeated here.

As shown in chapter 2.2 a variety of access control mechanisms exists, each suited for scenarios with differing degrees of administration efforts and dynamics. It was shown that in dynamic coalitions these scenarios correlate with the different degrees of membership dynamics, introduced in section 2.1.2. Consequently, the following sections propose a number of dynamic coalition models, with different access control mechanisms according to their degree of membership dynamics , i.e. *identity-based access control (*IBAC*)* for coalitions with low membership dynamics, *role-based access control* (*RBAC*) for medium membership dynamics and *attribute-based access control* (*ABAC*) as well as *trust-based access control* (*TBAC*) for dynamic coalitions with high membership dynamics.

Common for all the models of this dimension is the extension of the basic dynamic coalition model with the enumerated sets *Effect* and *Action* with the former providing a set of possible access control answers and the latter enlisting the possible actions on information to be requested:

```
enum Effect = {permit, deny, notapplicable}
enum Action = {write,read,copy,delete}
```

All further extensions and adoptions which are characteristic of the access control mechanism under consideration will be presented in the following sections of this chapter.

3.3 Coalitions with Low Membership Dynamics and IBAC

In section 2.2.1 it was argued that *identity-based access control (*IBAC*)* is well suited to implement access control in dynamic coalition scenarios with low membership dynamics. In order to model the enforcing of privacy for dynamic coalitions with low membership dynamics the basic dynamic coalition model from section 3.1 has to be extended or modified to meet the following additional model requirements:

Model Requirements

Structural requirements:

- Need to represent policies as sets of atomic access control rules. Each agent upon creation is to be equipped with an empty policy. The rule effects of a policy's rules are to be combined through a policy combining algorithm (e.g. permit-overrides, deny-overrides) which itself has to be defined by the model.
- Need to represent access control rules, which are to be associated with rule targets consisting of subjects (i.e. agents), objects (i.e. information item), actions (e.g. read, write) and a rule effect (permit, deny) stating if a request triple with a corresponding target is to be permitted or denied

Operational requirements:

- Need for a method which compares a request target to a rule target and returns the rule effect if the targets match and *notapplicable* if not.
- Need for methods for the evaluation of policies by combining the evaluated effects of every contained rule according to the policy combining algorithm.
- Need for methods to set up new rules or change existing rules. Deletion or deactivation of rules will then be represented through setting a rule effect to *notapplicable*.
- Need to adapt the methods for sharing and requesting information such as that every share of an information item with a coalition results in a sharing of the owning agent's policy to the coalition and that every request is first evaluated through the access control methods before being granted.

The names and the structure of these access-control components refer closely to the *OASIS* standard *XACML*. Certain aspects like the fact that *XACML*-policies may be combined to policy-sets and may have targets themselves are not mirrored in our model. Those features might be useful in certain scenarios and but are not essential for the basic concept of *IBAC*. However, the implementation or the adding of these *OASIS XACML* features is straightforward. Since the model is intended to be as abstract as possible it refrains from modeling these features.

3.3.1 DCs with IBAC - Signature

The first universes to consider are *Policy* and *Rules*[5] with the former supposed to be a collection of the latter. *Rules* apply to certain access requests and give information about how to proceed with them. Every *Agent* carries a *Policy*.

[5]Not to be confused with *ASM* rules.

```
universe Policy
universe Rule

function PolicyRules: Policy -> SET /*of Rule*/
function AgentPolicy: Agent -> Policy
```

A rule has a *Target*, being itself a triple of sets of *Subjects* (agents requesting access), *Resources* (information to be accessed) and *Actions* (write, read, etc). Every *Rule* is attached to an *Effect* stating how the access request of a certain *target* should be handled, where *notapplicable* is the result of a rule target that does not match a request. Also every rule has a record of its originator being the administrator of that particular rule, which is important for the possibility of changes to the rule.

```
universe Target

enum Effect = {permit, deny, notapplicable}
enum Action = {write, read, delete, copy}

function RuleTarget: Rule -> Target
function RuleEffect: Rule -> Effect
function RuleAdmin: Rule -> Agents

function TargetSubjects: Target -> SET /*of Agent*/
function TargetResources: Target -> SET /*of Information*/
function TargetActions: Target -> SET /*of Action*/
```

The policy combining algorithm states how to proceed with access requests that apply to targets of two or more rules with different effects. It has to be set for each *Policy* upon initialization: Being atomic elements of the access control mechanism, the rules return the evaluated effects to the policy which are then combined to a single policy effect according to the policy combining algorithm. More policy combining algorithms than the proposed two may be found in the OASIS standard. However, attention is restricted to those two because they are the most commonly used.

```
enum CombAlg = {denyoverrides,permitoverrides}
function PolicyRuleCombAlg: Policy -> CombAlg
```

Upon initialization, every agent is equipped with an empty policy for the adding of access control rules (see *SetUpEmptyAgent()* in the operational part of the model). In case of a coalition this initially empty policy functions as a "sharing policy" containing all the access control rules for information that is shared with the coalition.

The resulting answer of a request evaluation is returned to the requester, allowing or denying him access. If a *notapplicable* is returned it is due to the fact that a fitting rule for the request has not yet been formulated. It is up to the designer of the architecture how to proceed: If he is *deny-biased* he will treat it as a *deny*-answer. Respectively he will treat it as a *permit*-answer if he is *permit-biased.*

3.3.2 DCs with IBAC - Operations

The described functionality explained in the previous sections is provided through the following *IBAC* operations (describing internal operations of the access control components, which are not to be called directly by agents) and dynamic coalition model methods (describing the possible behavior of dynamic coalitions and agents, such as sharing, requesting, etc. which make use of the *IBAC* operations):

IBAC operations

A rule *TargetMatch()* checks if the request target coincides with the rule target by forming the intersections of the subjects, resources and actions respectively and stating that none of those intersections should be empty in case of a match. However, the subject, object and action fields may be empty themselves, stating that this target should be applied for all subjects, objects or actions respectively. Therefore, the following method identifies all possible case combinations in a number of cases:

```
rule TargetMatch(t1,t2) = {
  if not(TargetSubjects(t2) = {}) and not(TargetResources(t2) = {})
  and not(TargetActions(t2) = {}) then
    result :=
      not(TargetSubjects(t1) intersect TargetSubjects(t2) = {})
      and not(TargetResources(t1) intersect TargetResources(t2) ={})
      and not(TargetActions(t1) intersect TargetActions(t2) = {})

  if not(TargetSubjects(t2) = {}) and not(TargetResources(t2) = {})
  and TargetActions(t2) = {} then
    result :=
      not(TargetSubjects(t1) intersect TargetSubjects(t2) = {})
      and not(TargetResources(t1) intersect TargetResources(t2) ={})

  if not(TargetSubjects(t2) = {}) and TargetResources(t2) = {}
  and not(TargetActions(t2) = {}) then
    result :=
      not(TargetSubjects(t1) intersect TargetSubjects(t2) = {})
      and not(TargetActions(t1) intersect TargetActions(t2) = {})

  if not(TargetSubjects(t2) = {}) and TargetResources(t2) = {}
  and TargetActions(t2) = {} then
    result :=
      not(TargetSubjects(t1) intersect TargetSubjects(t2) = {})

  if TargetSubjects(t2) = {} and not(TargetResources(t2) = {})
  and not(TargetActions(t2) = {}) then
    result :=
      not(TargetResources(t1) intersect TargetResources(t2) = {})
      and not(TargetActions(t1) intersect TargetActions(t2) = {})

  if TargetSubjects(t2) = {} and not(TargetResources(t2) = {})
  and TargetActions(t2) = {} then
    result :=
       not(TargetResources(t1) intersect TargetResources(t2) = {})

  if TargetSubjects(t2) = {} and TargetResources(t2) = {}
  and not(TargetActions(t2) = {}) then
    result :=
      not(TargetActions(t1) intersect TargetActions(t2) = {})

    if TargetSubjects(t2) = {} and TargetResources(t2) = {}
    and TargetActions(t2) = {} then
      result := true
}
```

The rule *EvaluateRule()* evaluates a rule against a request:

```
rule EvaluateRule(req,rul) = {
seq tm <- TargetMatch(req,RuleTarget(rul)) next
  if RuleTarget(rul) = undef then result:= RuleEffect(rul)
  else if tm then result := RuleEffect(rul) else
    result:=notapplicable
}
```

The result of the *TargetMatch()*-method has first to be assigned as the value of the local variable *tm* before it may be considered in the following condition. The value of the variable may then be checked and if targets match *EvaluateRule* returns the effect from the associated rule or a *notapplicable* answer is provided in case of a non-match.

A policy is evaluated by forwarding the request to all matching rules after checking which combining algorithm to use.

```
rule EvaluatePolicy(req,pol) = {
  if PolicyRuleCombAlg(pol) = denyoverrides then
    seq e <- EvalRulesDO(req, PolicyRules(pol)) next result:=e
  if PolicyRuleCombAlg(pol) = permitoverrides then
    seq e <- EvalRulesPO(req, PolicyRules(pol)) result:=e
}

rule EvalRulesPO(req, ruleset) = {
seqblock
  temprules := ruleset
  effect := notapplicable
  while not(temprules= {}) choose r in temprules do {
    seq e <- EvaluateRule(req,r) next
      if e = permit then
        effect:= permit
      else if e = deny and effect = notapplicable then
        effect:=deny
    remove r from temprules
  }
  result := effect
endseqblock
}

rule EvalRulesDO(req, ruleset) = {
seqblock
  temprules := ruleset
  effect := notapplicable
  while not(temprules= {}) choose r in temprules do {
    seq e <- EvaluateRule(req,r) next
      if e = deny then
        effect:= deny
      else if e = permit and effect = notapplicable then
        effect:=permit
    remove r from temprules
  }
  result := effect
endseqblock
}
```

The *while*-construct in the evaluation methods iterates all the rules in the rule-set to be checked and combines the effect of each rule to a single effect, with *permit*-answers or *deny*-answers overriding according to the chosen combining algorithm, *permit-overrides* or *deny-overrides* respectively.

With those operations it is possible to model the possible behavior of agents in dynamic coalitions with *IBAC* in the following section.

Dynamic Coalitions Operations with IBAC

The rule *SetUpEmptyAgent* from section 3.1.2 has to be extended such that not only the **Agents**-universe but also the *Policy*-universe gets extended with a new Element respectively and names are assigned to them according to the input parameters. The created agent carries no information and his policy is initially empty. Initially the rule combining algorithm is set to *permitoverrides* reflecting a design decision, due to the fact that every policy needs a combining algorithm. This decision could be made at another part of the *ASM*-program or could be passed to the rule as a parameter.

```
rule  SetUpEmptyAgent(agentname) = {
  extend Agents with a do
  extend Policy with p do {
    CallByName(agentname) := a
    Name(a) := agentname
    AgentInfo(a) := {}
    AgentPolicy(a) := p
    PolicyRuleCombAlg(p) := permitoverrides
    PolicyRules(p) := {}
    CoalitionMembers(a):={}
 }
}
```

An agent may set up new access control rules by calling the following method with the subjects of the access control consideration, the resources to be accessed, the actions to be performed on them and the effect the agent wishes to be enforced on the corresponding request. The new rule will have to be shared with all agents, with whom prior sharing of information took place:

```
rule SetUpACRule(ag,sub,res,act,eff) = {
  extend Target with tar do
  extend Rule with rul do {
    TargetSubjects(tar) := sub
    TargetResources(tar) := res
    TargetActions(tar) := act
    RuleTarget(rul) := tar
    RuleEffect(rul) := eff
    RuleAdmin(rul):=ag
    forall i in AgentInfo(ag) do
      forall a in SharedWith(i) do
        add rul to PolicyRules(AgentPolicy(a))
  }
}
```

Access control rules may be changed by the agent who created them. The *RuleAdmin()*-function asserts this, prohibiting for example the possibility for a dynamic coalition to change a coalition policy rule, that has not been created by the

coalition itself but by an collaborating agent. A change of this rule will happen through an update of the rule that corresponds to the target that the agents wishes to be associated with a new effect. In order to deactivate a rule the overriding effect may also be *notapplicable*.

```
rule ChangeACRule(ag,sub,res,act,eff) = {
  forall r in PolicyRules(AgentPolicy(ag)) with
  TargetSubjects(RuleTarget(r))=sub and
  TargetResources(RuleTarget(r))=res and
  TargetActions(RuleTarget(r))=act do
    if RuleAdmin(r)=ag then RuleEffect(r):=eff
}
```

Agents may share information and may request access to information. The rule *ShareInfo()* of the basic DC-Model in the previous section has to be updated such that as soon as an information item is shared with a coalition, the current policy of the sharing agent is shared with the coalition also:

```
rule ShareInfo(ag,co,i) = {
  if InfoOwner(i)=ag then
    if exists a in CoalitionMembers(co) with a= ag then {
      forall rul in PolicyRules(AgentPolicy(ag)) do
        add rul to PolicyRules(AgentPolicy(co))
      add info to AgentInfo(co)
      add co to SharedWith(i)
    {
}
```

Requests to access information and how to handle them is essential in access control. The request-response process is modeled as follows: Coalition members who want to request access to certain information items call the rule *RequestInfo()*. First the request is translated into a *Target* and then it is forwarded to the part of the coalition policy for which the rules belong to the owner of the requested information. The resulting partial policy is then evaluated against the request and the effect returned in the above mentioned manner. It has to be noted that rules are designed so as that the originator of an information initially always has full access control privileges. Also, as in the basic dynamic coalition model agents may only access information if they are part of the coalition in the first place and *RequestInfo()* returns *notapplicable* if the requested resource cannot be found in the coalitions information repository.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result:=permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r in AgentInfo(co) with r=res then
        extend Target with tar do
        extend Policy with temppolicy do
        seqblock
          TargetSubjects(tar) := {ag}
          TargetResources(tar) := {res}
          TargetActions(tar) := {act}
          PolicyRules(TempPolicy):={rules | rules in
                                    PolicyRules(AgentPolicy(co)) with
                                   RuleAdmin(rules) = InfoOwner(res)}
          PolicyRuleCombAlg(TempPolicy):=
```

```
          PolicyRuleCombAlg(AgentPolicy(co))
          e <- EvaluatePolicy(tar, TempPolicy)
          if e = permit then
            print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          if e = deny then
            print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          if e = notapplicable then
            print Name(co)+ ": ACCESS FOR " +Name(ag)+ " NOT
                 APPLICABLE!"
          result:=e
        endseqblock
      else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
           +Name(res)+" NOT IN COALITION REPOSITORY!"
      next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
         +Name(ag)+" NOT IN COALITION!" next result:=notapplicable
}
```

Furthermore it is essential for the *ASM*-program to define that any kind of access may only take place if and only if the *RequestInfo()* returns *permit*. The fulfillment of this requirement will be demonstrated in the examples at the end of this chapter.

The rules *CreateInfo()*, *DeleteInfo()*, *AddCoalitionMembers()* and *RemoveCoalitionMembers()* remain unchanged from the basic dynamic coalition model.

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share information items with *identity-based access control*. Accesses to information items inside the coalition are then controlled according to the policy rules of the agents from whom the information originated. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

3.4 Coalitions with Medium Membership Dynamics and RBAC

In section 2.2.2 it was argued that *role-based access control* (*RBAC*) is well suited to implement access control in dynamic coalition scenarios with medium membership dynamics. The names and the structure of the access-control components presented here refer closely to the *NIST* standard *RBAC*. Since this standard offers a variety of role-based access control concepts in form of various models (see section 2.2.2) the following section will introduce a number of models with each one extending the previous one by adding or modifying functionality. The models shown are:

- *Flat RBAC*
- *Hierarchical RBAC*
- *Constrained RBAC*: *Static Separation of Duty (SSOD)*
- *Constrained RBAC*: *Dynamic Separation of Duty (DSOD)*

Since each *RBAC*-concept builds upon the former, each section will only consider the changes and extensions from and to the previous model, with an exception for

the two *constrained RBAC* variations which both built upon the *hierarchical RBAC* model.

In order to model the enforcing of *role-based access control* in its various forms in dynamic coalitions with medium membership dynamics the basic dynamic coalition model from section 3.1 has to be extended or modified to meet the following model requirements:

Model Requirements

Structural requirements:

- Need to represent roles and permissions which may be assigned to each other. Agents will then be assigned to roles thereby gaining access permissions to information items. Permissions are to consist of an information item and an action to be permitted on the former.
- Need to identify agents as role administrators who may then maintain the user-role and permission-role assignments.
- Need to represent a role hierarchy for *hierarchical RBAC* as a binary relation between two roles thereby adding to possibility to inherit permissions through the role hierarchy.
- Need to represent static *SSOD* or dynamic *DSOD* separation of duty constraints for *constrained RBAC*.
- Need to represent sessions and role activation for the application of dynamic separation of duty *DSOD* constraints for *constrained RBAC*.

Operational requirements:

- Need for methods for setting up and deleting roles and permission as well as assigning roles to users and permissions to roles. These methods may only be used by role administrators.
- Need for methods for creating new hierarchy relations in between roles as well as static or dynamic separation of duty constraints for *hierarchical RBAC* or *constrained RBAC* respectively. These methods may only be used by role administrators.
- Need to adapt the methods for sharing and requesting information such as that every share of an information item with a coalition results in a sharing of the owning agent's permissions which are concerned with that information item and that every request is first evaluated according to the role structure, the role hierarchies, the *SSOD* and *DSOD* constraints for *flat RBAC*, *hierarchical RBAC* and *constrained RBAC* respectively, before being granted.

First, for each model the extended signature is shown, followed by the definition of the *ASM* rules employed by the particular access control mechanism and finally the updated dynamic coalition methods will be presented which will then make use of the afore defined access control *ASM* rules.

3.4.1 DCs with Flat RBAC - Signature

The following universes need to be added to the basic dynamic coalition model, in order to set the background for the flat role-based access control components. *Permission*-elements will later be assigned to *Role*-elements.

```
universe Role
universe Permission
```

Roles get assigned to agents, who may carry several roles at a time. Also there is a function *RoleUsers* which records which agents have been associated with which role.

```
function UserRoles : Agents -> SET /*of Roles*/
function RoleUsers : Role -> SET /*of Agents*/
```

Initially a set of agents has to be equipped with administrative rights for creating roles and assigning them to users. This reflects the fact, that *RBAC* has to be administered by a set of agents and is therefore not completely suitable for highly dynamic distributed scenarios, in which there might be no time for planning or agreeing on who the administrators are to be. This assignment will be recorded in the following function, which will later be checked when calling the methods for creating and assigning roles.

```
function RoleAdmins: -> SET /*of Agents*/
```

Every permission will carry an ID based on the resource (information) and the action to be allowed, in order to make the addressing of certain permissions during runtime possible.

```
function PermissionID : Information * Action -> Permission
```

A permission is mapped to an information item it corresponds to as well as to the action that is to be permitted for this information item.

```
function PermissionRessource: Permission -> Information
function PermissionAction: Permission -> Action
```

Permissions may then be assigned to roles. Every role may carry a set of permissions, which as shown above, through the *RoleAgent()*-assignment may be indirectly assigned to agents.

```
function RolePermissions : Role -> SET /*of Permission*/
function PermissionRoles : Permission -> SET /*of Roles*/
```

3.4.2 DCs with Flat RBAC - Operations

The functionality for *flat RBAC* is provided through the following *flat RBAC* operations (describing operations concerning the management of the access control components) and the extended dynamic coalition model methods (describing the behavior of dynamic coalitions and agents, such as sharing, requesting, etc. which will also make use of the *flat RBAC* operations):

Flat RBAC operations

Essential for *flat RBAC* is the creation and management of roles. The following method creates a new role, initially with no assigned permissions and no assigned users. The creation, as well as the management and assignment of roles, should only be possible to be carried out by administrating agents, enforced by the first if-condition in both operations. *SetUpRole()* creates a new element of the *Role*-Universe for which the assigned permissions and users are initially empty:

```
rule SetUpRole(ag,rolename) = {
  if exists a in RoleAdmin with a=ag then
    extend Role with r do {
      CallByName(rolename) := r
      RolePermissions(r) := {}
      RoleUsers(r) := {}
   }
}
```

Deactivation of roles means getting rid of all assignments to or from that particular role:

```
rule  RemoveRole(ag,ro)= {
  if exists a in RoleAdmins with a=ag then {
    seq forall ag in RoleUsers(ro) do
      remove ro from UserRoles(ag)
    next RoleUsers(ro):={}
  }
}
```

Roles may be assigned or de-assigned to or from agents through calling of *AssignRoles()* or *DeAssignRoles()* respectively. The execution of the former will not only extend the set of roles which a user holds but also extend the set of agents which are actually assigned to this particular role with the latter removing them in the analogue way.

```
rule AssignRole(agent,ag,ro) = {
  if exists a in RoleAdmins with a=agent then {
    add ro to UserRoles(ag)
    add ag to RoleUsers(ro)
  }
}

rule DeAssignRole(agent,ag,ro) = {
  if exists a in RoleAdmins with a=agent then {
    add ro to UserRoles(ag)
    add ag to RoleUsers(ro)
 }
}
```

SetUpPermission() is called by an agent *ag*, who wants to set up a permission for an action *act* on a resource *res* that he owns. The *PermissionID* is assigned to the latter two and the initial roles associated with the newly created permission are empty. If the permission does not already exist, the new permission will be added to the set of permissions that the agent holds as well as to those of all other agents, with whom that particular information item already has been shared before. Note that only an information owner may use this method:

```
rule SetUpPermission(ag,res,act) = {
  if InfoOwner(res)=ag then
    if not exists p in Permission
    with PermissionID(res,act)=p then
      extend Permission with p do {
        PermissionRessource(p) := res
        PermissionAction(p) := act
        PermissionID(res,act) := p
        PermissionRoles(p) := {}
      }
}
```

Permissions may be assigned or de-assigned to agents through calling of *AssignPermission()* or *DeAssignPermission()* respectively. The execution of the former will not only extend the set of roles which a permission holds but also extend the set of permissions which are actually assigned to this particular role with the latter removing them in the analogue way.

```
rule AssignPermission(ag,ro,pe) = {
```

```
  if InfoOwner(PermissionRessource(pe))=ag then {
    add pe to RolePermissions(ro)
    add ro to PermissionRoles(pe)
 }
}

rule DeAssignPermission(ag,ro,pe) = {
  if InfoOwner(PermissionRessource(pe))=ag then {
    remove pe to RolePermissions(ro)
    remove ro to PermissionRoles(pe)
 }
}
```

Deactivation of permissions means getting rid of all assignments to or from that particular permission, as well as deleting the permissions stored with all the agents with whom the corresponding information item has been shared with. Note that only an information owner may deactivate corresponding conditions.

```
rule  DeactivatePermission(ag,pe)= {
  if InfoOwner(PermissionRessource(pe))=ag then {
    forall ro in PermissionRoles(pe) do
      remove pe from RolePermissions(ro)
    PermissionRoles(role):={}
 }
}
```

Dynamic Coalition Operations with Flat RBAC

The following dynamic coalition methods have to be extended and modified from the basic dynamic coalition model in order to integrate the *flat RBAC* mechanisms.

When setting up an empty agent, his sets of roles and permission are initially empty:

```
rule SetUpEmptyAgent(agentname) = {
  extend Agents with a do {
    CallByName(agentname) := a
    Name(a) := agentname
    AgentInfo(a) := {}
    UserRoles(a) := {}
    CoalitionMembers(a):={}
 }
}
```

Then the *RequestInfo()*-rule returns *true* or *false*, when the requesting agent holds roles with permissions for the requested action on the requested resource. Note that if the requesting agent is the owner of the requested information the access will always be granted. Also, as in the basic dynamic coalition model agents may only access information if they are part of the coalition in the first place and *RequestInfo()* returns *notapplicable* if the requested resource cannot be found in the coalitions information repository.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result := permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r in AgentInfo(co) with r=res then
```

```
        if exists p in Permission
        with PermissionRessource(p)=res and PermissionAction(p)=act
        and not(PermissionRoles(p) intersect UserRoles(ag) = {})
        then
          seq print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          next result := permit
        else
          seq print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          next result := deny
      else
        seq print Name(co)+ ": ACCESS NOT APPLICABLE!
            "+Name(res)+" NOT IN COALITION REPOSITORY!"
        next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
         +Name(ag)+" NOT IN COALITION!"
    next result:=notapplicable
}
```

The rules *CreateInfo()*, *DeleteInfo()*, *ShareInfo()*, *AddCoalitionMembers()* and *RemoveCoalitionMembers()* remain unchanged from the basic DC-model presented in section 3.1.

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share their information items with a *flat role-based access control* according to the permissions of the agents whom the information originated from. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

3.4.3 DCs with Hierarchical RBAC - Signature

The only addition to the signature of section 3.4.1 in order to integrate role hierarchies into the access control consideration is a function for recording role hierarchies. Role hierarchies are recorded in the boolean relation *RoleHierarchy()* which for two roles *r1* and *r2* points to *true* if and only if *r1* stands above *r2* in hierarchy, i.e. *r1* is to inherit all permissions that *r2* holds.

```
function RoleHierarchy : Role * Role -> BOOLEAN
```

3.4.4 DCs with Hierarchical RBAC - Operations

The functionality for *hierarchical RBAC* is provided through the following *hierarchical RBAC* operations (describing operations concerning the management of the access control components) and the extended dynamic coalition model methods (describing the behavior of dynamic coalitions and agents, such as requesting, etc. which will also make use of the *hierarchical RBAC* operations):

Hierarchical RBAC operations

Hierarchy relations may be added with the following method which defines *ro1* to be a senior role to *ro2*. After every adding of a role, the transitivity of the hierarchy relation is not guaranteed. Therefore a transitive closure has to take place in the last lines of this operation.

```
rule AddHierarchy(ag,ro1,ro2) = {
  if exists a in RoleAdmin with a=ag then
  seqblock
    RoleHierarchy(ro1,ro2) := true
    i:=1
    iterate
      if i<=size(Role) then {
        seq forall r1 in Role do
          forall r2 in Role do
            forall r3 in Role do
              if RoleHierarchy(r1,r2) and RoleHierarchy(r2,r3)
                then RoleHierarchy(r1,r3):= true
         next i:= i+1
      }
  endseqblock
}
```

Removing a hierarchy relation is straightforward. One should keep in mind that the hierarchy relations that have been created during the transitive closure will not be removed by this method as it might be desirable that they are maintained. Thus, a removal has to be made through call of the following function for every single relation at a time:

```
rule RemoveHierarchy(ag,ro1,ro2) = {
  if exists a in RoleAdmin with a=ag then
    Hierarchy(ro1,ro2):=false
}
```

Deactivation of roles means getting rid of all assignments to or from that particular role and deleting all hierarchy relations to or from that role:

```
rule RemoveRole(ag,ro)= {
  if exists a in RoleAdmin with a=ag then {
    seq forall a in RoleUsers(ro) do
      remove ro from UserRoles(a)
    next RoleUsers(ro):={}
    forall r in Role do {
      RemoveHierarchy(ag,ro,r)
      RemoveHierarchy(ag,r,ro)
    }
  }
}
```

Dynamic Coalition Operations with Hierarchical RBAC

Only the *RequestInfo()*-method has to be extended and modified from the *flat RBAC* model in section 3.4.1 in order to integrate the *hierarchical RBAC* mechanisms into the dynamic coalition operations.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result := permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r in AgentInfo(co) with r = res then
      seqblock
        temproles:= UserRoles(ag)
```

```
        forall ur in UserRoles(ag) do
          forall r in Role do
            if RoleHierarchy(ur,r)=true then add r to temproles
        if exists p in Permission  with
        PermissionRessource(p)=res and PermissionAction(p)=act
        and not(PermissionRoles(p) intersect temproles = {}) then
          seq print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          next result := permit
        else
          seq print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          next result := deny
      endseqblock
      else
        seq print Name(co)+ ": ACCESS NOT APPLICABLE! "+Name(res)+"
             NOT IN COALITION REPOSITORY!"
        next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
         +Name(ag)+" NOT IN COALITION!"
      next result:=notapplicable
}
```

All other dynamic coalition operations remain unchanged from the *flat RBAC* operations presented in section 3.1.2.

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share their information items with a *hierarchical role-based access control* mechanism for the information items inside the coalition to be enforced according to the permissions of the agents whom the information originated from. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

3.4.5 DCs with Constrained RBAC (SSOD)- Signature

The only addition to the signature of section 3.4.3 in order to integrate *constrained RBAC* is a function *SSOD()*. It records for each role which roles are to be excluded when being assigned to that role.

```
function SSOD : Role -> SET /*of Roles*/
```

3.4.6 DCs with Constrained RBAC (SSOD) - Operations

The functionality for *constrained RBAC (SSOD)* is provided through the following *constrained RBAC (SSOD)* operations (describing operations concerning the management of the access control components). The dynamic coalition model methods (describing the behavior of dynamic coalitions and agents such as sharing, requesting, etc.) remain unchanged from the *hierarchical RBAC* operations presented in section 3.4.3 .

Constrained RBAC (SSOD) operations

The *SetUpRole()* operation for *constrained RBAC (SSOD)* differs from its *flat RBAC* pendant only in the initialization of the *SSOD()* function for the role to

be created. The set of *static separation of duty constraints* for a newly created role is initially empty.

```
rule SetUpRole(ag,rolename) = {
  if exists a in RoleAdmin with a=ag then
    extend Role with r do {
      CallByName(rolename) := r
      RolePermissions(r) := {}
      RoleUsers(r) := {}
      SSOD(r):={}
   }
}
```

Removing of roles means getting rid of all assignments to or from that particular role and deleting all hierarchy relations to or from that role as well as all associated *static separation of duty* constraints:

```
rule RemoveRole(ag,ro)= {
  if exists a in RoleAdmin with a=ag then {
    seq forall a in RoleUsers(ro) do
      remove ro from UserRoles(a)
    next RoleUsers(ro) := {}
    forall r in Role do {
      RemoveHierarchy(ag,role,r)
      RemoveHierarchy(ag,r,role)
      remove ro from SSOD(r)
    {
    SSOD(ro):={}
  }
}
```

Roles may be assigned or deassigned to agents through calling of *AssignRoles()* or *DeAssignRoles()* respectively. The execution of the former will not only extend the set or roles which a user holds but also extend the set of agents which are actually assigned to this particular role with the latter removing them in an analogue way. However, before a role may be assigned, it has first to be checked if a *static separation of duty* constraint applies to this situation. This could be the case because either the agent directly holds a role that is supposed to be excluded from the assignment of the new role or if this is the case for one of his inherited roles with the latter reflecting the inheritance of *SSOD* constraints throughout a hierarchy.

```
rule AssignRole(agent, ag, ro) = {
  if exists a in RoleAdmin with a=agent then
  seqblock
    temproles := UserRoles(ag)
    forall ur in UserRoles(ag)  do
      forall r  in Role  do
        if RoleHierarchy(ur,r) = true then add r to temproles
    if temproles  intersect SSOD(ro) = {}  then {
      add ro to UserRoles(ag)
      add ag to RoleUsers(ro)
    }
    else print "ERROR: Cannot assign role due to SSOD constraints!"
  endseqblock
}
```

Hierarchy relations may me added with the following rule which defines *ro1* to be a senior role to *ro2*. However hierarchies may only be assigned when the relation to be created does not violate the pre-existing *SSOD* constraints: The two roles may only be put into a hierarchy relation when the two roles are not subject to *SSOD* constraints. After every adding of a role, the transitivity of the hierarchy relation is not guaranteed. Therefore, a transitive closure has to take place in the last lines of this operation:

```
rule AddHierarchy(ag,ro1,ro2) = {
  if exists a in RoleAdmin with a=ag then
    if exists r in SSOD(ro1) with r=ro2
    or exists r in Role with
    SSOD(ro1) intersect {r}={r} and RoleHierarchy(ro2,r)     or
    SSOD(r) intersect {ro2} = {ro2}  and RoleHierarchy(r,ro1) then
      print "ERROR: Can't create Hierarchy due to SSOD constraints!"
    else
    seqblock
      RoleHierarchy(ro1,ro2) := true
      i:=1
      iterate
        if i<=size(Role) then {
          seq forall r1 in Role do
            forall r2 in Role do
              forall r3 in Role do
                if RoleHierarchy(r1,r2) and RoleHierarchy(r2,r3)
                  then RoleHierarchy(r1,r3):= true
          next i:= i+1
        }
    endseqblock
}
```

To set up a static *separation of duty constraint* the method *SetUpConstraint()* may be called by a role administrator with the roles that are supposed to exclude each other. The constraint will be enforced during the assignment of roles from then on. However, the constraint will only be recorded if and only if no prior existing hierarchy is in conflict with the constraint. If that was to be the case the hierarchy would have to be changed first before the constraint may be recorded.

```
rule SetUpConstraint(ag,ro1,ro2) ={
  if exists a in RoleAdmin with a=ag then
    if RoleHierarchy(ro1,ro2) = false
    and RoleHierarchy(ro2,ro1) = false then {
      add ro1 to SSOD(ro2)
      add ro2 to SSOD(ro1)
    }
    else print "ERROR: Can't set up constraint due to
         existing Hierarchy!"
}
```

Dynamic Coalition Operations with Constrained RBAC (SSOD)

All dynamic coalition operations remain unchanged from the *hierarchical RBAC* model presented in section 3.4.4.

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the

structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share their information items with a *constrained role-based access control* mechanism for the information items inside the coalition to be enforced according to the permissions of the agents whom the information originated from, while considering *dynamic separation of duty* constraints for the role assignment. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

3.4.7 DCs with Constrained RBAC (DSOD)- Signature

The only two additions to the signature of section 3.4.3 in order to integrate *constrained RBAC (DSOD)* are the two functions *DSOD()* and *Session()*. A function *DSOD()* records for each rule which roles are to be excluded when being activated to that role.

```
function DSOD : Role -> SET /*of Roles*/
```

Furthermore a function *Session()* records sessions. Here the concept of a session is abstracted to the aspect which is essential for enforcing *dynamic separation of duty* constraints: the activation of roles. Agents may activate roles they are assigned to, as long as the activation does not violate the *DSOD* policy , i.e. agents may be assigned to conflicting roles, as long as they are not activated at the same time. To that end the function *Session()* records which roles are currently activated for an agent:

```
function Session : Agents -> SET /*of Roles*/
```

3.4.8 DCs with Constrained RBAC (DSOD) - Operations

The functionality for *constrained RBAC (DSOD)* is provided through the following *constrained RBAC (DSOD)* operations (describing operations concerning the management of the access control components) and the extended dynamic coalition model methods (describing the behavior of dynamic coalitions and agents, such as requesting, etc.) which will also make use of the *constrained RBAC (DSOD)* operations):

Constrained RBAC (DSOD) Operations

When setting up an empty agent, his sets of roles and permission are initially empty, as well as the activate roles in his Session:

```
rule SetUpEmptyAgent(agentname) = {
  extend Agents with a do {
    CallByName(agentname) := a
    Name(a) := agentname
    AgentInfo(a) := {}
    UserRoles(a) := {}
    CoalitionMembers(a):={}
    Session(a):={}
 }
}
```

The *SetUpRole()* operation for *constrained RBAC (DSOD)* differs from the its *flat RBAC* pendant only in the initialization of the *DSOD()* function for the role to be created. The set of *dynamic separation of duty constraints* for a newly created role is initially empty.

```
rule SetUpRole(ag,rolename) = {
  if exists a in RoleAdmin with a=ag then
    extend Role with r do {
      CallByName(rolename) := r
      RolePermissions(r) := {}
      RoleUsers(r) := {}
      DSOD(r):={}
   }
}
```

Removing of roles means getting rid of all assignments to or from that particular role and deleting all hierarchy relations to or from that role as well as all associated *dynamic separation of duty* constraints:

```
rule RemoveRole(ag,ro)= {
  if exists a in RoleAdmin with a=ag then {
    seq forall a in RoleUsers(ro) do
      remove ro from UserRoles(a)
    next RoleUsers(ro) := {}
    forall r in Role do {
      RemoveHierarchy(ag,ro,r)
      RemoveHierarchy(ag,r,ro)
      remove ro from DSOD{r}
    {
    DSOD(ro):={}
  }
}
```

In *RBAC* with *DSOD* only activated roles in the particular session matter for the evaluation of access. A previously assigned role, as well as the inherited roles of the assigned roles, may be activated through the following method which first checks if the role to-be-activated is actually assigned to the user and if yes activates the role, if and only if no *DSOD* constraint restricts this particular activation in the current session.

```
rule ActivateRole(ag,ro) = {
  seqblock
    temproles := UserRoles(ag)
    forall ur in UserRoles(ag)  do
      forall r  in Role  do
        if RoleHierarchy(ur,r)=true  then add r  to temproles
    if temproles intersect {ro} = {} then
      print "ERROR: Cannot activate unassigned role!"
    else
      if Session(ag) intersect DSOD(ro) = {} then
        add ro to Session(ag)
      else print "ERROR: Cannot activate role due to SSOD
           constraints!"
  endseqblock
}
```

Roles may by deactivated in the obvious manner:

```
rule DeActivateRole(ag,ro) = {
  remove ro from Session(ag)
}
```

To set up a *DSOD* constraint the method *SetUpConstraint()* may be called by a role administrator with the roles that are supposed to exclude each other. The constraint will from then on be enforced during the activation of roles. Note that these constraints are recorded independently from prior existing role hierarchies, because unlike as in *static separation of duty* the constraints they have only to be considered upon activation.

```
rule SetUpConstraint(ag,ro1,ro2) = {
  if exists a in RoleAdmin with a=ag then {
    add ro1 to DSOD(ro2)
    add ro2 to DSOD(ro1)
  }
}
```

Dynamic Coalition Operations with Constrained RBAC (DSOD)

Only the *RequestInfo()*-method has to be extended and modified from the *hierarchical RBAC* model in section 3.4.3 in order to integrate the *constrained RBAC (DSOD)* mechanisms. The *RequestInfo()*-method returns *true* or *false*, when the requesting agent holds roles with permissions for the requested action on the requested item. This may be validated by checking if any of the activated roles in this particular session provide permissions for this request. Therefore a temporary variable *temproles* is created and initialized with all the user's roles as well as the roles he inherits. Note that if the requesting agent is the owner of the requested information the access will always be granted.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result := permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r  in AgentInfo(co)  with r=res then
        if Session(ag) = {} then
          seq print Name(co)+ "ACCESS DENIED! No roles activated!"
          next result := false
        else if exists p in Permission  with
        PermissionRessource(p)=res and PermissionAction(p)=act
        and not(PermissionRoles(p) intersect  Session(ag) = {})
        then
          seq print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          next result := permit
        else
          seq print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          next result := deny
      else
        seq print Name(co)+ ": ACCESS NOT APPLICABLE! "+Name(res)+"
          NOT IN COALITION REPOSITORY!"
        next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
         +Name(ag)+" NOT IN COALITION!"
    next result:=notapplicable
}
```

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share their

information items with a *constrained role-based access control* mechanism for the information items inside the coalition to be enforced according to the permissions of the agents whom the information originated from, while considering *dynamic separation of duty* constraints for the role assignment. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

3.5 DCs with High Membership Dynamics and ABAC

In section 2.2.3 it was argued that *attribute-based access control* (*ABAC*) is well suited to implement access control in dynamic coalition scenarios with high membership dynamics. In order to model the enforcing of privacy for dynamic coalitions with high membership dynamics the basic dynamic coalition model from section 3.1 has to be extended with several universes, functions and rules in order to adequately model the access control components for *(ABAC)*.

The names and the structure of *ABAC* components in the proposed model refer closely to the OASIS standard *XACML* which aside from *identity-based access control* (*IBAC*) also allows for the definition of *ABAC* policies. Therefore the model components of the *ABAC* model resemble strongly those of the *IBAC* model in section 3.3, with some significant extensions and changes to integrate the reasoning over attributes. Therefore, the next sections will only show the changes made to the *IBAC* model for the purposes of integrating *ABAC*. All unchanged operations will not be shown again here, and the reader is referred to sections 3.3.1 and 3.3.2 for the complete picture.

In order to model the enforcing of *attribute-based access control* policies in dynamic coalitions with high membership dynamics the basic dynamic coalition model from section 3.1 has to be extended or modified to meet the following model requirements:

Model Requirements

Structural requirements:

- Need to represent policies as sets of atomic access control rules. Each agent upon creation is to be equipped with an empty policy. The rule effects of a policy's rules are to be combined through a policy combining algorithm (e.g. permit-overrides, deny-overrides) which itself has to be defined by the model.
- Need to represent attributes, which be may be created before or during runtime. Attributes have a type (e.g. role, name, age etc.) and a value.
- Need to represent access control rules, which are to be associated with a rule target consisting of free number of attributes.

Operational requirements:

- Need for a method which compares a request target to a rule target and returns the rule effect, if the targets match and *notapplicable* if not.
- Need for methods for the evaluation of policies by combining the evaluated effects of every contained rule according to the policy combining algorithm.
- Need for methods to set up new rules or change existing rules. Deletion or deactivation of rules will then be represented through setting a rule effect to *notapplicable*.
- Need for methods to create new attributes which may then be assigned to agents or to access control rules.
- Need to adapt the methods for sharing and requesting information such as that every share of an information item with a coalition results in a sharing of the owning agent's policy to the coalition and that every request is first evaluated through the access control methods before being granted.

3.5.1 DC with ABAC - Signature

The signature for dynamic coalitions with *IBAC* from section 3.3.1 has to be extended with the *Attribute* universe. Attributes determine the applicability of a rule to an access request. They might be dynamically (during runtime) created by agents. Attributes have an attribute types (defined by a string name) in order to make the attribute values (string values at this point) comparable. Attributes are associated with agents and with information-items, being the subject and the resource of a request respectively.

```
universe Attribute

function AttributeType: Attribute -> STRING
function AttributeValue: Attribute -> STRING

function SubjectAttributes : Agents -> SET
function ResourceAttributes: Information -> SET
```

Here a completely free attribute structure was chosen, meaning that the possible attribute types are not restricted to a predefined set. This brings up a major dilemma of applying *ABAC* in dynamic coalition scenarios:

Excursus: ABAC Discussion

With no restriction on attribute types and therefore with no prerequisites as to which attributes every collaborating agent has to provide, it may be cumbersome to guarantee that all the agents that are supposed to have access do actually know which attributes they need to provide

in order gain this access. Thus, in practice a common set of attributes is usually defined for each scenario. This set could be imposed on coalition members as a prerequisite for the collaboration or it could be reached through mutual agreement between the agents. The outcome of these approaches could be defined in model terms, such as that the possible attribute types are not completely free but a fixed enumerable set of predefined entities. The attribute part of the signature for an exemplary attribute structure would have to change as follows:

```
universe Attribute

function AttributeType: Attribute -> AttributeTypes
function AttributeValue: Attribute -> STRING

enum AttributeTypes: {age,job_position,job_role}
```

Here, only the predefined attribute types, such as the three in this example, may be used for policy definition and utilized in access requests. This second, predefining approach may be necessary in most scenarios in order to ensure the work of a dynamic coalition by avoiding the inherent problems of the completely open *ABAC* approach. However, this approach also looses one of the core ideas and major advantages of ABAC: the ability to deal with *relatively* (with exception of their attributes) unknown agents. In short the dilemma at hand is as follows: How free or how well defined do the attributes structures have to be in order to guarantee a) that all agents that are supposed to have access, have access and b) the main advantage of *ABAC*, i.e. the ability to deal with relatively unknown agents, is not lost.

For the presentation of the *ABAC* model in this section it was decided to stick with the completely open approach, first because despite its shortcomings in actual application, it reflects the idea of *ABAC* more naturally and second because it shifts the modeling focus not on the necessary consensus but on the actual mechanism needed to deal with the attributes. However, without loss of generality, in later case studies a predefined set of attribute types may be easily defined such as in the example above.

A universe *Condition* is needed in order to create conditional access rights. A *Condition* is defined by a type *ConditionType* which maps a condition to a string, which identifies the variable type (e.g. name, role, age, etc.) in question and is supposed to be of a certain value *ConditionValue* according to a match function *ConditionMatchID* (e.g. "=" or "¡"). *ConditionValue* may be an *Element* like a string, a boolean or a number, while *ConditionMatchID* has to remain a string for comparison purposes.

```
universe Condition

function ConditionType : Condition -> STRING
function ConditionValue: Condition -> ELEMENT
function ConditionMatchID: Condition -> STRING
```

A rule has a *RuleTarget*. However, different from *IBAC*, the structure of this target is not predefined (and therefore the universe *Target* not needed) but is merely a set of attributes: *attributes* of the *subjects* (agents requesting access), the *resources*

(information to be accessed),the *actions* (write, read, etc) or of possible other entities which may be defined by the modeler such as the environment (time, location, etc.). All attributes in a *RuleTarget* are optional, i.e. a *RuleTarget* may be empty, resulting in a application of the *RuleEffect* for every request.

```
function RuleTarget: Rule -> SET
```

3.5.2 DCs with ABAC - Operations

The described functionality explained in the previous sections is provided through the following ABAC operations (describing internal operations of the access control components, which are not to be called directly by agents) and dynamic coalition model operations (describing the behavior of dynamic coalitions and agents, such as sharing, requesting, which will also make use of the ABAC operations). Remember that only those operations which have to be newly created or modified from the *IBAC* model in section 3.3.2 are shown here:

ABAC Operations

A rule *TargetMatch* checks if the request target coincides with the rule's conditions target by checking if the attributes required by the rule are provided. This check happens according to a matching function *ConditionMatchID()* which is a string of the sort "=" or "¡" and more. How many matching ID's are supported is up to the modeler. For reasons of space limitations, here only two matching functions are explicitly shown. Further ones may be added in an analogue way. If all attributes match *true* is returned and if at least one attribute does not match *false* respectively:

```
rule TargetMatch(t1,t2) = {
  seqblock
    nomatch:={}
    forall cond in t2 do {
      /* "="-match-comparison */
      if ConditionMatchID(cond) = "=" then
        if not(exists a in t1
        with ConditionType(cond)=AttributeType(a)) then
          add false to nomatch
        else choose a in t1
        with ConditionType(cond)=AttributeType(a) do
          if not(ConditionValue(cond)=AttributeValue(a)) then
       add false to nomatch
/* ">"-match-comparison */
      if ConditionMatchID(cond) = ">" then
        if not(exists a in t1
        with ConditionType(cond)=AttributeType(a)) then
          add false to nomatch
        else choose a in t1
        with ConditionType(cond)=AttributeType(a) do
          if not(ConditionValue(cond)<AttributeValue(a)) then
      add false to nomatch
 /* define more match comparisons here */
    }
    if nomatch={} then result:=true
    else result:=false
  endseqblock
}
```

The rest of the evaluation operations for the *ABAC* model remain unchanged from the *IBAC* mode. Together with those operations it is possible to model the possible behavior of agents in dynamic coalitions with *ABAC* in the following section.

Dynamic Coalitions Operations with ABAC

A rule for the creation of attributes has to be introduced. Who is able to call this rule is to be defined for every use case at a time. Some attributes are most likely automatically created, such as attributes for the identity of an agent (see *SetUpEmptyAgent()* further down). Other attributes could be created by an administrative *ASM* agent, who records certain attributes for every collaborating agent. The rule *CreateAttribute()* returns an attribute of the type and value according to the submitted parameters.

```
rule CreateAttribute(type,value)= {
  if not exists a in Attribute with AttributeType(a)=type
  and AttributeValue(a)=value then
    extend Attribute with attr do
    seqblock
      AttributeType(attr):=type
      AttributeValue(attr):=value
      result:=attr
    endseqblock
}
```

Attributes for subjects and resources may be set up through the call of the following rules. A new attribute gets created only if no attribute of this particular type already exists for the subject or the resource respectively. If an attribute of the type exists, its value will be overwritten with the new value.

```
rule SetSubjectAttribute(type,value,agent)= {
  if exists b in SubjectAttributes(agent)
  with AttributeType(b) = type then
    choose c in SubjectAttributes(agent)
    with AttributeType(c) = type do
      AttributeValue(c) := value
    else
    seqblock
      at <- CreateAttribute(type,value)
      add at to SubjectAttributes(agent)
    endseqblock
}

rule SetResourceAttribute(type,value,agent)= {
  if exists b in ResourceAttributes(res)
  with AttributeType(b) = type then
    choose c in ResourceAttributes(res)
    with AttributeType(c) = type do
      AttributeValue(c) := value
    else
    seqblock
      at <- CreateAttribute(type,value)
      add at to SubjectAttributes(agent)
```

```
    endseqblock
}
```

The rule *SetUpEmptyAgent()* from section 3.3.2 has to be extended so that an initial attribute for the agent's identity is created and added to the set of the agent's attributes.

```
rule  SetUpEmptyAgent(agentname) = {
  extend Agents with a do
  extend Policy with p do {
    CallByName(agentname) := a
    Name(a) := agentname
    AgentInfo(a) := {}
    AgentPolicy(a) := p
    SubjectAttributes(a):={}
    SetSubjectAttribute("Subject",a,a)
    PolicyRuleCombAlg(p) := permitoverrides
    PolicyRules(p) := {}
    CoalitionMembers(a):={}
 }
}
```

Rule Conditions may be created through call of the rule *CreateRuleCondition()*. As shown above a condition has a type, a value and a matching function, which is to be used in the comparison part of the *TargetMatch*-rule. *CreateRuleCondition()* returns the newly created condition. This condition may later be used (among others) to create an access control rule.

```
rule CreateRuleCondition(type,value,matchID) = {
  extend Condition with c do
  seqblock
    ConditionType(c) := type
    ConditionValue(c) := value
    ConditionMatchID(c) := matchID
    result:=c
  endseqblock
}
```

An agent may set up new access control rules by calling the following method with the conditions of the request the rule is supposed to correspond to. The conditions may relate to attributes of the subject of the access control consideration, the resource it wants to access or may state the action it wants to perform on it. The condition set may also be empty, stating that the corresponding effect is to be returned for every request. Thus, the *SetUpACRule()* rule from 3.3.2 has to be changed as follows:

```
rule SetUpACRule(ag,cond,eff) = {
  extend Rule with rule do {
    RuleTarget(rule) := cond
    RuleEffect(rule) := eff
    RuleAdmin(rule):= ag
    forall res in AgentInfo(ag) do
      forall a in SharedWith(res) do
        add rule to PolicyRules(AgentPolicy(a))
 }
}
```

Access control rules may be changed by the agent who created them. The *RuleAdmin()*-function assures this, for example prohibiting the possibility for a dynamic coalition to change a coalition policy rule that has not been created by the coalition but by an collaborating agent. Different from the corresponding rule in the *IBAC* model a change of the a rule will happen through an update of the rule that corresponds to the condition set that the agent wishes to be associated with a new effect. In order to deactivate a rule, the overriding effect may also be *notapplicable*.

```
rule ChangeACRule(ag,cond,eff) = {
  seq temprules := PolicyRules(AgentPolicy(ag))
  next while (not(temprules= {}))
    choose r in temprules do {
      if RuleAdmin(r)=Ag then
        seq if RuleTarget(r)=Cond then RuleEffect(r):=Eff
        next remove r from temprules
   }
}
```

The *CreateInfo()*-method has to be extended so that an attribute for the resources ID is set.

```
rule CreateInfo(agent,infoname) = {
  extend Information with i do {
    CallByName(infoname) := i
    Name(i):=infoname
    InfoOwner(i):=agent
    SharedWith(i):={agent}
    ResourceAttributes(i):={}
    SetResourceAttribute("Resource",i,i)
    add i to  AgentInfo(agent)
 }
}
```

Information items may be removed from an agent's repository. However, only the information owner is allowed to remove his information, either from his own repository or from a coalitions repository that he shared his information with. By deleting an information item from an information set of an agent all rules corresponding to that particular information item will also be deleted.

```
rule DeleteInfo(ag1, ag2, i)= {
  if InfoOwner(i)=ag1 then {
    remove i from AgentInfo(ag2)
    forall r in PolicyRules(AgentPolicy(ag2)) do
      if exists cond in RuleTarget(r) with
      AttributeType(cond)="Resource" and
      AttributeValue(cond)=i then
        remove r from PolicyRules(AgentPolicy(ag2))
 }
}
```

The request-response process for *ABAC* differs from its *IBAC* equivalent in the following way: Coalition members who want to request access to certain information items call the rule *RequestInfo()* with their own ID, the ID of the coalition they want to access, the resource they want to access and the type of access they want

to perform. First the request is translated into a set of attributes consisting of the request attributes for the subject, the resource and the action. A temporary policy is created which consists of all policy rules that belong to the owner of the information which is to be accessed. [6] The resulting attribute set is then evaluated for this temporary policy, which then returns the associated effect.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result:=permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r  in AgentInfo(co)  with r=res then
        extend Policy  with tempPol  do
        seqblock
          a <- CreateAttribute("Action",act)
          RequestAttributes:= {a} union ResourceAttributes(res)
                                  union SubjectAttributes(ag)
          PolicyRules(tempPol):={rules | rules in
                                 PolicyRules(AgentPolicy(co)) with
                                 RuleAdmin(rules) = InfoOwner(res)}
          PolicyRuleCombAlg(tempPol) :=
               PolicyRuleCombAlg(AgentPolicy(co))
          e <- EvaluatePolicy(RequestAttributes, tempPol)
          if e = permit then
            print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          if e = deny then
            print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          if e = notapplicable then
            print Name(co)+ ": ACCESS FOR " +Name(ag)+ " NOT
                 APPLICABLE!"
          result:=e
        endseqblock
      else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
           +Name(res)+" NOT IN COALITION REPOSITORY!"
        next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
        +Name(ag)+" NOT IN COALITION!"
    next result:=notapplicable
}
```

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share information items with *attribute-based access control*. Accesses to information items inside the coalition are then controlled according to the policy rules of the agents from whom the information originated. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

[6] This is necessary because rule targets may be empty and may therefore apply to every request. In a coalition this fact has to be restricted that only the resource owner's policies are evaluated in order to avoid the interference with the other agents' policies.

3.6 DCs with High Membership Dynamics and TBAC

In section 2.2.4 it has been argued that *trust-based access control* (*TBAC*) is well suited to implement access control in dynamic coalition scenarios with high membership dynamics, meaning coalitions in which little to no knowledge of an agent's policy format, role or attribute structure exists prior to the collaboration. In order to model the enforcing of privacy for dynamic coalitions with such high membership dynamics the basic dynamic coalition model from section 3.1 has to be extended with several universes, functions and rules in order to adequately model the access control components for *TBAC*. As shown in 2.2.4 the basic concept of *TBAC* utilized for this thesis closely refers to the *NIST* standard *RBAC* with the main difference that when speaking about trust one does intuitively mean the trust of one agent in another agent. Hence usually each agents carries his own "trust list" being a mapping of trust values to all agents known to him as opposed to the role-based concept, where role definitions take place on a global level and are the same for all coalition members. In order to remain true to this intuitive approach, the proposed work will model the components in conformance with this idea. However, in an excursus it will be shown that trust on a global level may be easily represented with very few modifications to the model e.g. considering an agents trust in a coalition as a whole before access control consideration. Nevertheless, many of the *flat RBAC* components presented in section 3.4.1 are identical to the ones used here. Therefore, only the ones that are newly created or modified from its *flat RBAC* equivalent are presented here.

In order to model the enforcing of *TBAC* policies in dynamic coalitions with high membership dynamics the basic dynamic coalition model from section 3.1 has to be extended or modified to meet the following model requirements.:

Model Requirements

Structural requirements:

- Need to represent permissions which may be assigned to a number representing a trust degree, thereby stating that only agents that are trusted to this degree or more are to gain this permission.
- Need to represent the trust of one agent in another. If another agent is unknown a default trust value is to be set. Every agent has his own trust assignment list.

Operational requirements:

- Need for methods for setting up or deleting permission as well as assigning permissions to trust values and trust values to agents (the latter could also happen automatically through invocation of certain methods or actions which are to be defined).
- Need to adapt the methods for sharing and requesting information such as that every share of an information item with a coalition results in a sharing of the owning agent's policy, i.e. his permissions and trust requirements.

3.6.1 DCs with TBAC - Signature

The *flat RBAC* signature from section 3.4.1 has to be extended with a function *TrustReq()* which assigns permissions to trust degrees being themselves numbers, preferably a rational one between 0 and 1. The assigned trust degree is seen as the required trust for allowing access for a requesting agent.

```
function TrustReq: Permission -> NUMBER
```

Trust of one agent in another is recorded as a function mapping of two agents to a number (usually a rational number between 0 and 1). This function states

that the first input agent trusts the second input agent to the assigned trust value degree.

```
function Trust: Agents * Agents -> NUMBER
```

A function *DefaultTrust* defines the default value every agent is assigned to by the other agents as soon as he enters a coalition. The default value may be changed and modified before as well as during runtime according to the modeling decisions or scenario requirements respectively. In this case the default value is defined to be 0.

```
function  DefaultTrust:  -> NUMBER initially 0
```

3.6.2 DCs with TBAC - Operations

The functionality for *flat RBAC* is provided through the following *TBAC* operations (describing operations concerning the management of the access control components) and the extended dynamic coalition model methods (describing the behavior of dynamic coalitions and agents, such as sharing, requesting, etc. which will also make use of the *TBAC* operations):

TBAC Operations

Essential for the implementation of *trust-based access control* is the maintenance of an agents access control list. Trust may change over time. This change may be invoked manually by an agent or in an automatic fashion, depending of the trust evaluation concept applied. Since the various ways and concepts through which trust values may be evaluated and changed over time are not considered in this model, the change of a trust value is summarized in the following simple rule which may be called by an agent himself or could be invoked by another "evaluation"-agent, who changes the trust values at a given period of time or after certain actions.

```
rule ChangeTrust(ag1,ag2,trustvalue) = {
  Trust(ag1,ag2):=trustvalue
{
```

SetUpPermission() is called by an agent *ag*, who wants to set up a permission for an action *act* on a resource *res* that he owns. The *PermissionID* is correlated with the latter two and the initial trust requirement is maximal in order to only allow fully trusted individuals this access. If the permission does not already exist, the new permission will be added to the set of permissions that the agent holds as well as to those of all other agents with whom that particular information item has been shared already. Note that only an information owner may use this method:

```
rule SetUpPermission(ag,res,act) = {
  if InfoOwner(res)=ag then
    if not exists p in AgentPermissions(ag)
    with PermissionID(res,act)=p then
      extend Permission with p do {
        PermissionRessource(p) := res
        PermissionAction(p) := act
        PermissionID(res,act) := p
        TrustReq(p):=1.0
        forall a in SharedWith(res) do
          add p to AgentPermissions(a)
    }
}
```

The trust requirement of a permission may be set by the owner of the resource that the permission corresponds to through the following rule:

```
rule  SetTrustReq(ag,pe,trustdegree)= {
  if InfoOwner(PermissionRessource(pe))=ag then
    TrustReq(p):=trustdegree
}
```

A rule *CalculateTrust()* is responsible for the evaluation of a new trust degree according to given parameters. Since the definition of this rule varies from use case to use case, it will not be defined here but left open for definition for actual scenarios, such as in the second case study of this thesis in section 4.1.3.

```
rule CalculateTrust(ag1, ag2) = {
/*to be defined by the modeler*/
}
```

A rule *ChangeTrustParam()* changes an agent trust parameter that is subject to the trust evaluation to a new value. After the change the trust in the agent is re-evaluated:

```
rule ChangeTrustParam(ag,param,value) = {
  seqblock
    param:=value
    forall co in Agents
    with {ag} intersect CoalitionMembers(co) = {ag} do
      CalculateTrust(co,ag)
  endseqblock
}
```

Dynamic Coalition Operations with TBAC

The following dynamic coalition operations have to be extended and modified from the *flat RBAC* model presented in section 3.4.1 in order to integrate the *TBAC* mechanisms.

When an agent joins a coalition, the coalition or the members of the coalition will evaluate this trust. How this evaluation takes place is dependent of the *CalbulateTrust()* rule, which may be defined differently for different use cases. There, also the definition of the rule *AddCoalMember()* will vary. Supposing that a default trust value is to be assigned to an arriving agent, the adding of agents to coalition has to slightly be modified in the following way: All agents that do not know the newly arrived agent yet (i.e. have not worked with him in a prior coalition) will assign the default trust degree to an agent who joins a coalition for the first time, which serves as the a marker for the initial trust a completely unknown agent enjoys amongst the other agents.

```
rule AddCoalMember(ag, co) = {
  forall a in CoalitionMembers(co)  do
    if Trust(a,ag) =  undef then Trust(a,ag):=DefaultTrust
  CoalitionMembers(co) := CoalitionMembers(co) union {ag}
}
```

Then the *RequestInfo()*-method returns *true* or *false* when the requesting agent is trusted by the information owner to the degree the permission specifies or more.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result := permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r in AgentInfo(co) with r = res then
        if exists p in AgentPermissions(co)  with
        PermissionRessource(p)=res and PermissionAction(p)=act
        and  Trust(InfoOwner(res),ag) >= TrustReq(p) then
          seq print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          next result := permit
        else
          seq print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          next result := deny
      else
       seq print Name(co)+ ": ACCESS NOT APPLICABLE!
            "+Name(res)+" NOT IN COALITION REPOSITORY!"
      next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
         +Name(ag)+" NOT IN COALITION!"
    next result:=notapplicable
}
```

Excursus: DCs with *TBAC* at Coalition Level - Operations

The model presented so far considers trust to be recorded for each agent individually as this approach models the intuitive understanding of trust more closely. However, it is also thinkable to consider trust at a coalition level, meaning that trust for each agent is recorded not by each agent in his personal store but by the coalition he takes part in. An access request will then take the agents trust from the coalition as a whole and determine, whether he may gain access or not.

The following methods have to be adapted from the previous *TBAC* model in order to model *trust-based access control* at coalition level.

The method for adding agents to coalitions will set a default value for the coalition's trust in the agent which is to be added rather than setting the default value for every agent's trust in the new agent such as in previously presented model:

```
rule AddCoalMember(ag, co) = {
  if Trust(co,ag) =  undef then Trust(co,ag):=DefaultTrust
  CoalitionMembers(co) := CoalitionMembers(co) union {ag}
}
```

The request of one agent to a coalition will then use the coalition's trust in the requesting agent and compare it to the trust requirements of the correlated permission.

```
rule RequestInfo(ag, co, res, act) = {
  if InfoOwner(res)=ag then result := permit else
    if exists a in CoalitionMembers(co) with a= ag then
      if exists r in AgentInfo(co) with r=res then {
        if exists p in AgentPermissions(co)  with
        PermissionRessource(p)=res and PermissionAction(p)=act
        and Trust(co,ag) >= TrustReq(p) then
          seq print Name(co)+ ": ACCESS GRANTED FOR " +Name(ag)+ "!"
          next result := permit
```

```
        else
          seq print Name(co)+ ": ACCESS DENIED FOR " +Name(ag)+ "!"
          next result := deny
      }
      else
        seq print Name(co)+ ": ACCESS NOT APPLICABLE!
             "+Name(res)+" NOT IN COALITION REPOSITORY!"
        next result:=notapplicable
    else seq print Name(co)+ ": ACCESS NOT APPLICABLE! "
         +Name(ag)+" NOT IN COALITION!"
    next result:=notapplicable
}
```

With these slight modifications the presented models may also be used to model *trust-based access control*, with the trust consideration taking place at coalition-level rather than in the previously defined model which considers trust at agent-level.

Together with the model components presented in this section the basic dynamic coalition model from section 3.1 gets extended and modified so that it fulfills all the structural and operational model requirements defined at the beginning of this section. It is now possible to model a dynamic coalition in which agents may share information items with *trust-based access control*. Accesses to information items inside the coalition are then controlled according to the policy rules of the agents from whom the information originated. Note that these policies get enforced even after the owning agent himself leaves the coalition, guaranteeing his privacy even after he is not collaborating anymore.

3.7 Applying the Framework

The proposed models may be used at various points of the software engineering life cycle. According to [BS03] The following activities may be seen as major for such a life cycle:

- **Requirement Capture**: Börger et al. propose the use of *ASM* ground models to capture the main requirements and characteristics of a software system. The dynamic coalition models in this thesis are such ground models, with the basic dynamic coalition model serving as the lowest level ground model, capturing at first only the process and the interactions that take place in between agents.
- **Detailed Design**: Further refinement of the ground models will allow for the modeling of more concrete scenarios, for example defining access control policies and thereby integrating non-quantitative quality aspects into the modeling. Thereby, as proposed in [BS03], a stepwise refinable abstract operational modeling as well as a model driven development approach become possible. Since every *ASM* provides the notion of an *ASM* run, for every abstraction level the ability to validate and verify is a given.
- **Validation**: *ASMs* provide the notion of *ASM* runs, being executions of *ASM* programs, thereby allowing for testing and simulation of the models. In order to bridge the gap between validation and verification, statistical tests may be performed: Tests for non-deterministic models may be tested a great

number of times, each time recording the adherence of the considered properties and finally making a quantitative statement about them such as: with a probability of 90% the *ASM* run will terminate. [7]

- **Verification**: Logical properties for *ASM* may be verified through manual proof techniques or by means of model checking techniques. The proposed dynamic coalitions model therefore may be seen as a first basis for the task of verifying dynamic coalitions scenarios specifications. Properties to verify could be merely logical, such as deadlock-freeness of safety but could also be operational, for example stating that no agent may leave a coalition before he has actually joined it. A number of model checking tools exist for various *ASM* tools including one solution for the *CoreASM* environment used in this work. For more information on how to use those tools with the proposed models see section 5.2.

- **Documentation**: The programming language independent structure of *ASM* definitions facilitates the readability of such models, to the point that it is used as a documentation tool with the aim to improve understandability of a software project, providing at the same time a unique understanding of the considered components and structures. In the case of dynamic coalitions scenarios it may be used as a common documentation language in which, independent of the agent's own software infrastructure, process and policies are defined globally as common agreement. A newly arriving member would then easily be able to understand the ongoing process and integrate itself in it according to the model. At the same time the simplicity of the models allow for a more effective integration of domain experts, who with relatively little effort may understand the models and then add their domain knowledge to the modeling process allowing for a model that is closer to the actual domain needs.

In the case studies in Chapter 4 it will be presented how the proposed model may be used for the mentioned software development activities in two real-life scenarios taken from the medical sector. Thereby, the thesis stated in section 1.3 will be validated. However, as already mentioned in the first chapter, the *verification*-phase will not be regarded in these case studies as the adoption of existing tools and model checkers for the verification of *CoreASM* programs is not in the scope of this work and therefore remains subject to future research.

3.7.1 Correctness of the Framework

Any application of the framework in real-life scenarios has to be preceded by a check of the correctness of the presented models. This correctness has been validated through validation by means of simulations, making sure that all above mentioned operations of the various models function in the expected ways, i.e. that they update the *ASM* state in a correct manner. Here, "correct" denotes the intuitive way in which the described dynamic coalition and access control operations are supposed to work according to their informal specification (see sections 2.1 and 2.2). Simulations were used from the beginning of the modeling process and for every model component, i.e. operations. For every newly created operation, simulations of its functionality have been conducted with the tools presented in the next section. For example, one simulation consisted of printing out the state locations which were to be modified by the operation in question before and after the operation was

[7] Note that in this thesis the validation of the models is solely conducted through simulation means, while the integration of testing methods is left for future work.

executed in an *ASM* program. Another utilized means to validate the correctness of model parts was the utilization of the *State Exploration GUI* in the next section, by which a complete *ASM* run may be investigated state by state thereby inferring if the operations work according to specification or not.

3.7.2 Tool Support for the Framework

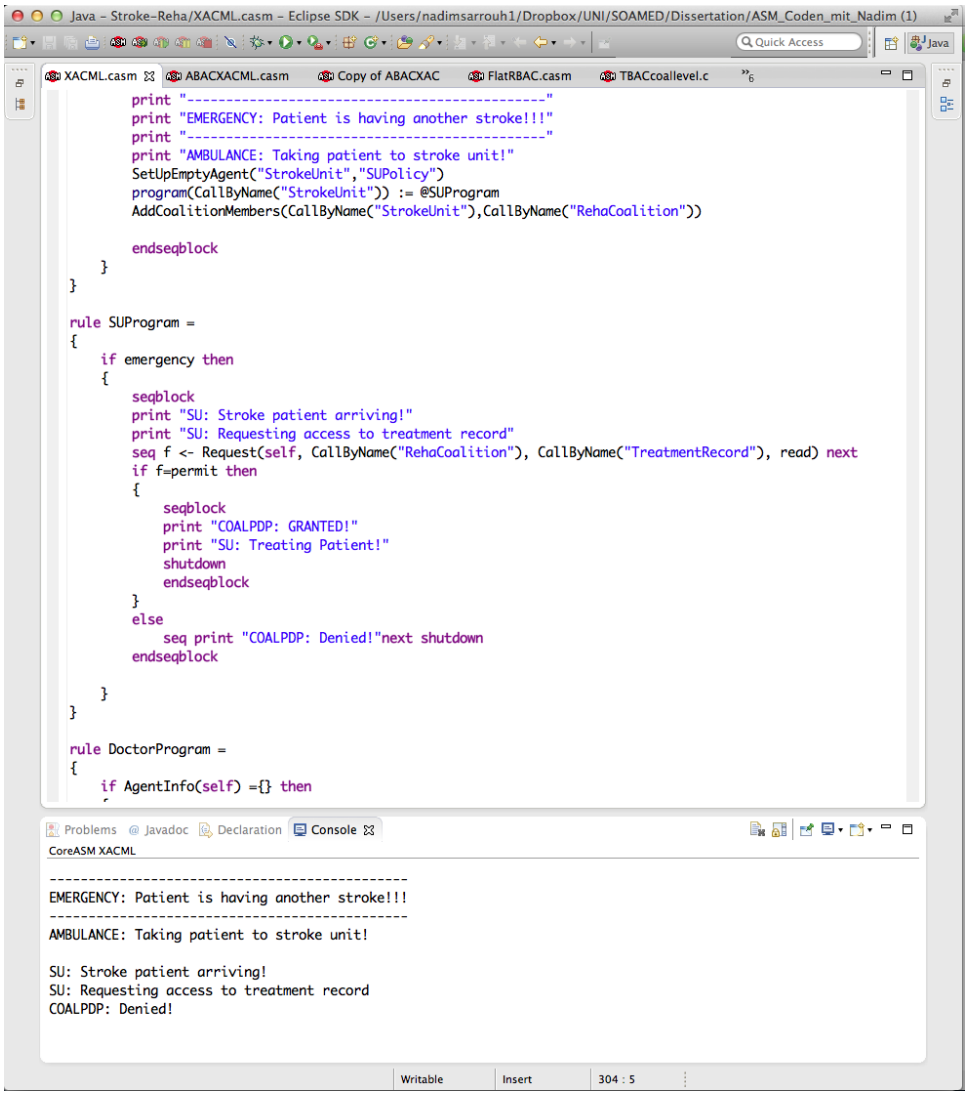

Figure 3.1: *CoreASM* simulation with command line outputs.

The *CoreASM Eclipse Plugin* (see figure 3.1) supports using the proposed framework of this work in conducting those five activities: It serves as a developing environment for *requirement capture* (ground model definition) as well as more *detailed designs*. It allows for the execution of the actual *ASM programs* in so called *ASM* runs, thereby providing *validation* means like tests and simulation. Through *input* and *output* rules the simulation is provided with an interactive element which allows the user to follow certain output lines as well as programming the *ASM* in such a way that its execution depends one certain input commands from the user. A *CoreASM-Model Checker* [FGM] allows for the automatic *verification* of logical

properties for *CoreASM* definition. Last but not least the created *CoreASM* definitions serve as an excellent means to document design decision by formalizing them for a unique and unambiguous understanding. Moreover an understandability of the *ASM* code helps to close the knowledge gap between software developers and domain experts.

In *CoreASM* the states of the *ASM* are represented internally and normally not visualized for the user. However the *Observer Plugin* provided for *CoreASM* allows to export predefined parts of the states of an *ASM* run to an XML-file. In the course of this thesis a tool has been created which uses this XML-file for a *state exploration GUI* (see figure 3.2) which allows to graphically browse an *ASM* run in order to gain a better understanding of the developments of universes and function definition. Thereby, one may for example investigate how policies evolve over time and why they return certain effect for access requests.

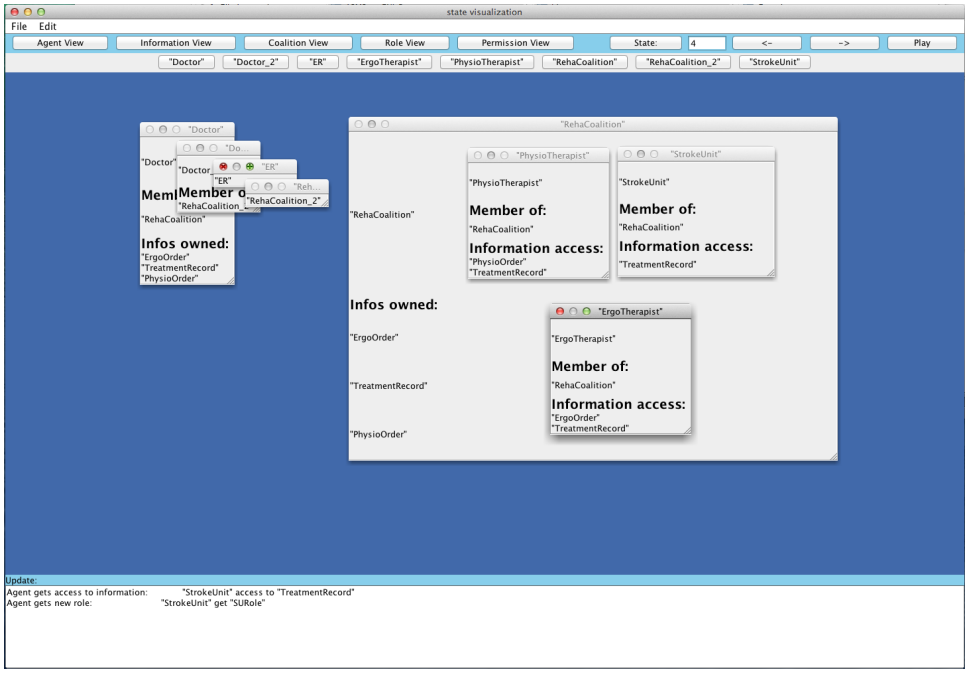

Figure 3.2: State exploration GUI.

The presented tools were not only used to model simulate models for the case studies, but were also equipped for experimental validation of the correctness of the framework itself.

3.7.3 An Example ASM program

The proposed models of the previous sections provide information about the structure of a dynamic coalition as well as a definition of the possible state transitions, representing the possible behavior of agents in a coalition (such as joining, leaving, sharing, etc.). An actual process may now be defined through the definition of an *ASM* program. The following example is a fictional one with close resemblance to processes that were observed at the *Charité Berlin* during the course of this thesis. The example makes use of the model for dynamic coalition with *IBAC* (see section 3.3). The example process that is recorded in the following section may be summarized as follows:

> After the rehabilitation process in a stationary clinic a stroke patient is discharged and the ongoing ambulant rehabilitation measures are now overlooked and managed by the patient's family doctor. This doctor receives information of the rehabilitation process in the clinic through an explicit doctor's letter, containing all the treatment up to this date as well as recommendations for further ambulant treatment. The doctor then decides what ambulant therapist the patient should see, e.g. physio-therapist, ergo-therapist, logo-therapist, etc. He sets up therapy orders and sends the patient to the therapists in question. After some therapy sessions the patient has another stroke and is delivered to the emergency room or stroke unit without any ability to speak or communicate. The emergency doctor now wants to access the information generated in the ambulant rehabilitation process in order to investigate what caused the stroke or just to understand what actually happened, thereby optimizing their emergency treatment.

This process and its actors can be seen as a typical dynamic coalition with "doctor", "stroke-unit", "physio-therapist", "ergo-therapist", etc. being the cooperating agents, sharing information about a stroke patient in order to treat him. Trying to support a process like this with a dynamic-coalition-architecture is one of the main motivations of the framework introduced in this thesis. Modeling a process with the presented framework happens through the specification of the single agent's programs (their behavior in the distributed *ASM*). A simulation of such a process is then provided through an actual *ASM* run of the total machine, being the union of all the agents programs executed in parallel. In the following section it will be demonstrated how to capture the proposed processes through *ASM*-rules and programs. Because of space limitations the therapist's programs will be omitted here because the actual internal processes of the therapy is not of interest for the scope of this model. Instead, the emergency-program which initiates an emergency at a random time, and the program of the stroke-unit, which then tries to access information of the treatment-process will be illustrated in detail.

The Init-Rule

In *CoreASM Init-Rule* is the first executed rule from where other agents such as the *Doctor* agent may be initialized. In the following code example an empty agent with the name "Doctor" is set up. Furthermore a coalition agent called *RehaCoalition* and an agent called *ER* are initialized. The latter is an *ASM* agent which models an emergency happening after a random amount of time (reflected through a global variable *emergency* initially set as false). The agent's programs are each set to a certain distributed program definition (a rule that has to be specified). The *Init-Rule* itself is not an agent, i.e. it runs once and only once at the beginning of the *ASM*-run.

```
rule initRule = {
  seqblock
    SetUpEmptyAgent("Doctor")
    program(CallByName("Doctor")) := @DoctorProgram

    SetUpEmptyAgent("RehaCoalition")
    program(CallByName("RehaCoalition")) := @coalProgram

    AddCoalitionMembers(CallByName("Doctor"),
         CallByName("RehaCoalition"))
```

```
    SetUpEmptyAgent("ER")
    program(CallByName("ER")) := @EmergencyProgram

    randomNumber := round(random() * 5)
    patientDischarged:=true
    ambulantTreatment:=false
    emergency:= false

    Agents(self) := false
    endseqblock
}
```

The DoctorProgram-Rule

The doctor's process is given in the agent program *DoctorProgram*. A new agent *PhysioTherapist* is created and linked to the corresponding *ASM* programm *PT-Program*. Furthermore, two information items are created: A treatment record file and a new order for the physio-therapist, with the corresponding access control rules which allow him access to the order as well as to the treatment record file.

```
rule DoctorProgram = {
  if patientDischarged = true and ambulantTreatment=false then
  seqblock
    ambulantTreatment:=true
    CreateInfo(self, "TreatmentRecord")

    SetUpEmptyAgent("PhysioTherapist")
    AddCoalitionMembers(CallByName("PhysioTherapist"),
    CallByName("RehaCoalition"))
    program(CallByName("PhysioTherapist")) := @PTProgram

    CreateInfo(self,PhysioOrder)
    SetUpACRule(self,{CallByName("PhysioTherapist")},
         {CallByName("PhysioOrder")},{read,write}, permit)
    SetUpACRule(self,{CallByName("PhysioTherapist")},
         {CallByName("TreatmentRecord")},{write}, permit)

    ShareInfo(self, CallByName("RehaCoalition"),
         CallByName("PhysioOrder"))
    ShareInfo(self, CallByName("RehaCoalition"),
         CallByName("TreatmentRecord"))
    /*Initiate more Therapists*/
  endseqblock
}
```

The EmergencyProgram-Rule

EmergencyProgram uses the *CoreASM*-variable *stepcount* to initiate an emergency after a random amount of time. It then terminates all other running agents and instantiates another agent which represents the stroke unit. Since the stroke unit now takes part in the treatment of the patient it is also added to the coalition so that it may try to access coalition information.

```
rule EmergencyProgram = {
  if stepcount  > randomNumber  and not emergency then
```

```
  seqblock
    terminate CallByName("Doctor")
    terminate CallByName("PhysioTherapist")
    terminate CallByName("ErgoTherapist")
    /*terminate other agents*/
    emergency := true
    SetUpEmptyAgent("StrokeUnit"),)
    program(CallByName("StrokeUnit")) := @StrokeUnitProgram
    AddCoalitionMembers(CallByName("StrokeUnit"),
    CallByName("RehaCoalition"))
  endseqblock
}
```

The StrokeUnit-Program

According to the predefined process the stroke unit tries to access treatment information in the coalition in order to continue the treatment. If the access is granted it starts the emergency treatment. If access is denied the *ASM* shuts down at this point, which suffices for the scope of this first example. Further actions for this case could be defined. For example the stroke unit could then try to contact the doctor and ask him to implement an access control rule which grants access to the stroke unit. Even better, access control rules of the like could be created in the beginning in order to prevent such bottlenecks or deadlocks.

```
rule StrokeUnitProgram = {
  if emergency then
  seqblock
    eff <- RequestInfo(self, RehaCoalition, TreatmentRecord, read)
    if eff=permit then
    /*  Start Emergency Treatment */
    else
      shutdown
  endseqblock
}
```

With simulation features of the *CoreASM*-tool it is possible to actually run these programs and eventually find out that the stroke unit may not work properly if neither policy-changes are made or the process itself is changed. The model helps to detect these bottlenecks or deadlocks and therefore serves as an excellent tool to model dynamic coalition scenarios with emphasized privacy and access control aspects before actually implementing an architecture to support those coalitions.

Chapter 4

Case Studies

In this chapter the applicability of the proposed framework for privacy-sensitive dynamic coalitions will be examined through two case studies. Furthermore the thesis of this work presented in section 1.3, will be validated against those two real-life scenarios. Both case studies are taken form the medical sector and have been created in joint work with institutions at the *Charité Berlin* which are cooperating with the *SOAMED Research Training Group* which this thesis is associated with.

Both case studies constitute scenarios which are regarded as dynamic coalitions of different membership dynamics with the requirement of privacy enforcing mechanisms. The thesis of this work states that the proposed framework for modeling privacy-sensitive dynamic coalitions supports the development of supporting software for these dynamic coalitions throughout the typical software engineering life cycle. Thus, the presented case studies aim to demonstrate how the framework may be utilized for *requirement capture, detailed design, validation* and *documentation*.

Case Study 1 examines an existing dynamic coalition in a stroke patient treatment process located at the *Charité Virchow Berlin* hospital. As this collaboration is dynamic in the sense that the members and the shared information depends on the patient and his parameters and needs, it obviously constitutes some kind of dynamic coalition. However, the process is well-defined (see the following BMPN models) and the collaborating agents are previously known as well as their possible interactions. Thus, it seems reasonable to interpret it as a dynamic coalition with low membership dynamics. Interestingly, during the course of this case study it becomes evident that viewing the coalition as one with medium membership dynamics is in fact more suitable.

The main purpose of the application of the model in this scenario is *requirement capture* and *detailed design*: With the basic dynamic coalition model (see section 3.1) the main process of the stroke patient treatment was formalized. This model is to be seen as the *requirement capture* of the process defining how the process should look like. This first, pure process model is to be seen as the ground model, defining the requirements of the process which are to be kept in later refinement steps. In a more *detailed design* access control policies according to the chosen access control mechanism were defined, thus beginning to enforce privacy-requirements. The adding of access control policies is not the only refinement to the model that would have to take place in order to make way for an actual software support of this coalition. It is merely the first needed refinement step in a number of steps which are to be iterated in the *detailed design* phase. However, because privacy and access control in medicine are so critical for the success of a software project, this refinement step should take place right after the *requirement capture* phase. The defined process is then simulated and validated against the defined policies and it is checked whether the policy definitions are adequate or need to

be adapted or changed to fulfill the process requirements, thereby addressing first *validation* issues. These simulations are conducted by the modeler alone but also in interaction with medical personnel associated with this process, demonstrating how the models provide a unique and easily conceivable understanding of the process and its members access control policies, setting up a foundation for discussion and improvement of them. On the other hand the formal model constitutes a first ground model for a possible future software project which is to support the dynamic coalitions in this scenario. Furthermore, the created *ASM* program may be utilized as an abstract specification *documentation* throughout the course of the software development project.

Case Study 2 deals with *requirement capture* and *detailed design* of a up to this day non-existent software platform for the exchange of research data of the *Newborn Hearing Screening* at *Charité Berlin* where examination and therapy information of the newborn patient is stored centrally. Access to an anonymized set of this information is frequently requested by various institutions all across Germany. To this day the granting of access rights has to be determined manually by a employee who will check the background and the credibility of a requesting agent through a time intensive research of his publications, reviews etc. This scenario constitutes a dynamic coalition with highly dynamic membership where previously unknown agents who contribute or want to access research data from the *Newborn Hearing Screening* form a dynamic coalition with the latter. One of the main requirements for a supporting software is to determine the access rights to research data for agents who are previously not or little known. Therefore, the application of the model for *attribute-based access control* or *trust-based access control* in highly dynamic coalitions is analyzed in this case study.

The presentation of both case studies is organized as follows. First an informal abstract scenario description is provided followed by *requirement capture* in form of a process formalization in the basic dynamic coalition model. In a more *detailed design* the motivation for the access control mechanism of choice is explained and the policy definition is formalized. A report on how the models were *validated* will follow as well as a description on how to use the created models for *documentation* purposes. Finally, this chapter ends with a summarizing conclusion as well as an account of the lessons learned from the application the framework presented in this thesis.

4.1 Stroke Treatment Process at Charité Berlin

During the course of the *SOAMED Research Training Group* the PhD-students have surveyed the stroke treatment process in the *Charité Virchow Berlin* hospital. This process is well defined and strictly timed as the timely treatment of a stroke patient may be vital for his survival and rehabilitation. As an outcome of this survey a process model in *BPMN* has been created together with medical experts from the hospital. This *BPMN* model will provide the basic understanding of the subject-matter of this case study. [1]

Since this hospital's stroke treatment process is well-defined it involves actors from one common institution (the hospital) and mostly works according to plan it seems safe to say that this scenario constitutes a dynamic coalition with low membership dynamics. Therefore, the possibility of modeling the access control requirements with *XACML* access control policies for *identity-based access control*

[1] On the next page, the complete *BPM* of the stroke treatment process is depicted for an overview of the process' complexity rather than a detailed presentation. Due to space limitations it is not clearly readable. For a more readable and detailed presentation of the model check the following pages or refer to the *PDF*-file on the CD attached to this thesis.

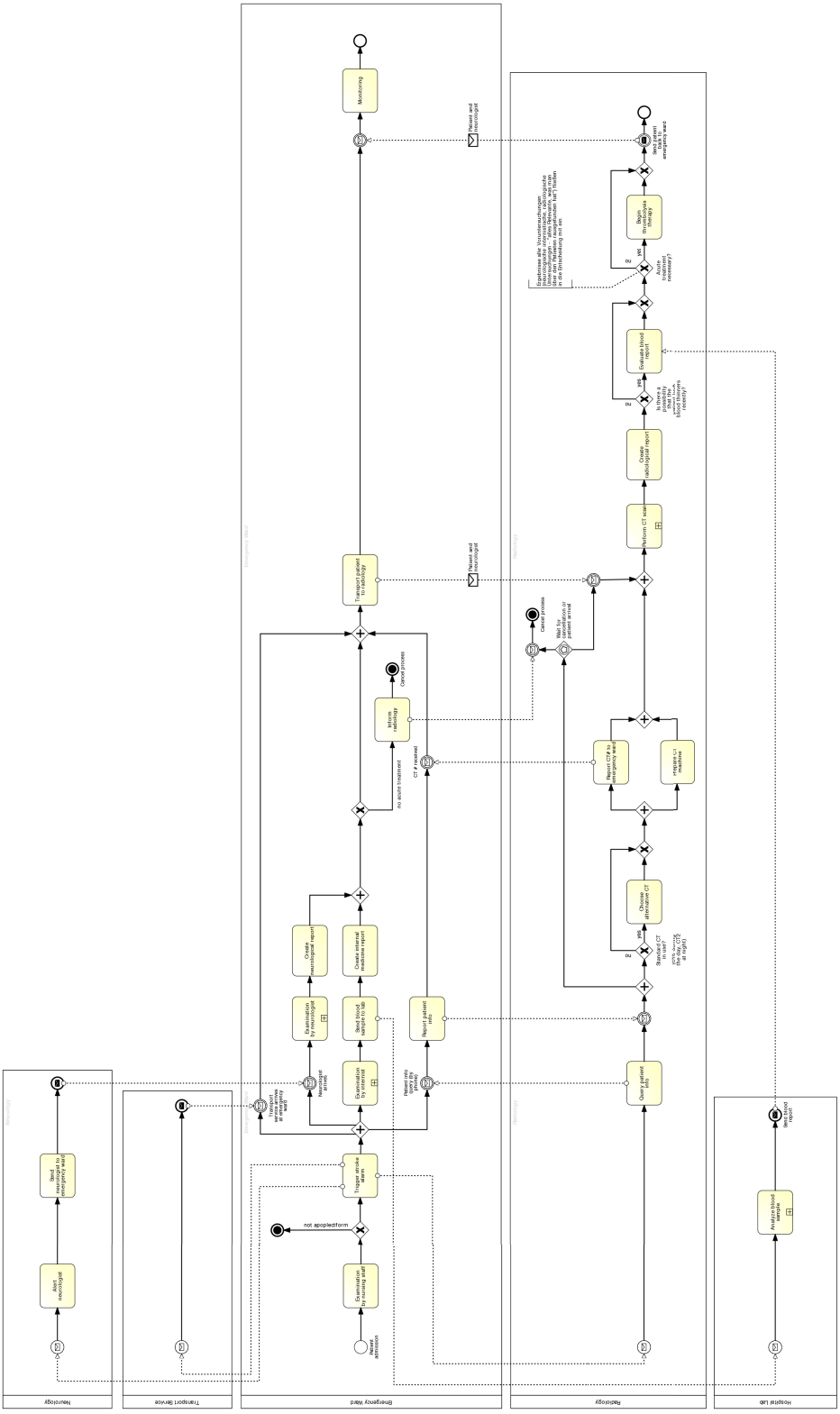

Figure 4.1: Overview of size and complexity of the complete stroke treatment process at *Charité Virchow Berlin* hospital.

(*IBAC*) will be investigated first. However, it will be shown how it may help to regard the scenario as a dynamic coalition with medium membership dynamics because some of the individuals cooperating in this scenario may indeed vary from case to case and therefore a role-based approach may be necessary.

Note that the scenario definition takes place on an very abstract level in form of a *ground model* which identifies the main components and activities of the coalition, and allows for a later refinement and detailed modeling of the vast amount of details that scenarios like these usually provide.

4.1.1 Informal Scenario Description

An informal description of the stroke patient treatment scenario at the *Charité Virchow Berlin* depicted in 4.1 could be as follows:

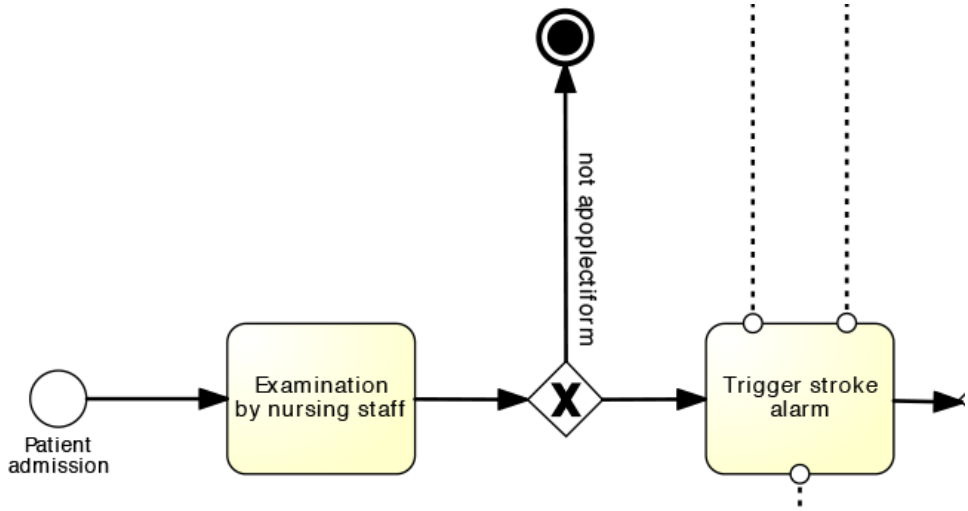

Figure 4.2: Emergency room: Initial Process with output messages to neurology, radiology and transport service.

A person *A* is delivered to the emergency room (see fig. 4.2). The nursing staff will conduct a first examination. In case the patient is apoplectiform (meaning that indicators of a stroke are observable) a stroke alarm is activated, sending messages and data to radiology, neurology and to a transport service which will later be needed for the transport of the patient. If the patient is not apolectiform this stroke process instance terminates and other treatment will be considered.

Neurology, radiology and the transport service notice these alarms. Radiology asks the emergency room via telephone for more patient information and holds its process until it is received (see fig. 4.3).

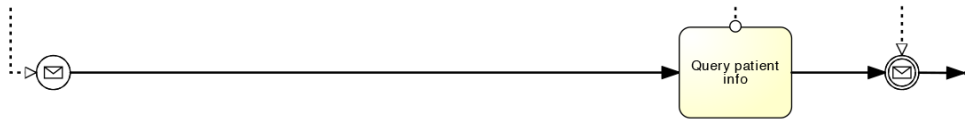

Figure 4.3: Radiology: Initial process extract after notification from emergency room, waiting for patient info to arrive as input from the emergency room.

Neurology immediately contacts a neurologist and sends him to the emergency room, terminating the neurology's involvement in this process (see fig. 4.4).

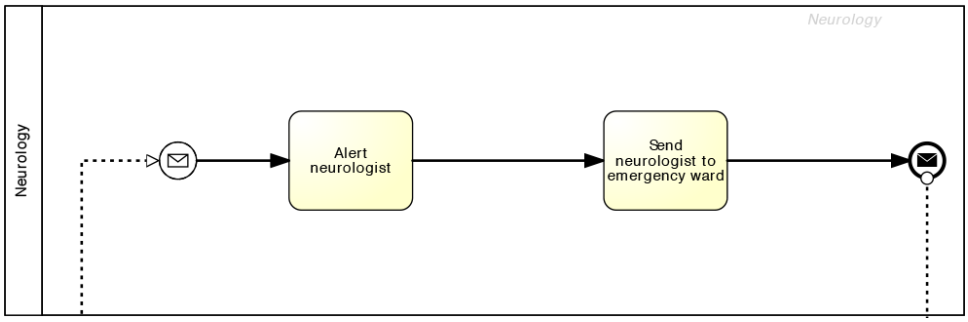

Figure 4.4: Neurology: Complete process after notification from emergency room, with output symbolizing the sending of a neurologist to the emergency room.

Transport service will do nothing more than actually send a transporter to the emergency room (see fig. 4.5).

Figure 4.5: Transport service: Complete process after notification from emergency room sending a transport service to the emergency room.

In the next step, the emergency room will start with parallel execution of a number of actions: First, after the arrival of the neurologist, a neurological examination of the patient will take place followed by the creation of a neurological report. Second, an internal examination takes place in parallel, resulting in the sending of a blood sample to the laboratory which will be involved with the coalition from then on. This path comes to a close with the creation of an internal report and together with the first path triggers the next process step. Parallel to those two paths the emergency room will receive the transport service and also send patient data to radiology. (see fig. 4.6).

After receiving the patient info from the emergency room radiology starts a parallel execution: in the first branch it waits for further info from the emergency room informing it of necessary acute therapy. The radiology's further process, such as the actual creation of a CT depends on the outcome of that evaluation. In another parallel step radiology will determine if the standard CT machine is available and if not choose another one. Then, it will prepare the CT and send a CT-number to the emergency room in parallel. From then on radiology will be waiting for further notice from the emergency room (see fig. 4.7).

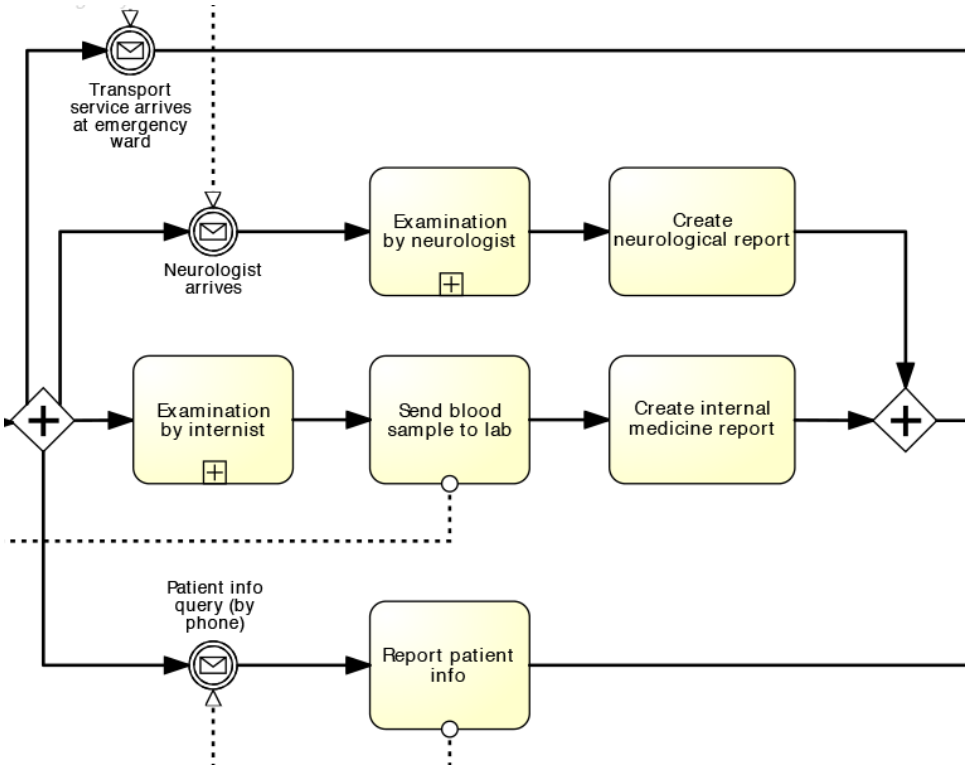

Figure 4.6: Emergency room: Process after the initial alarm.

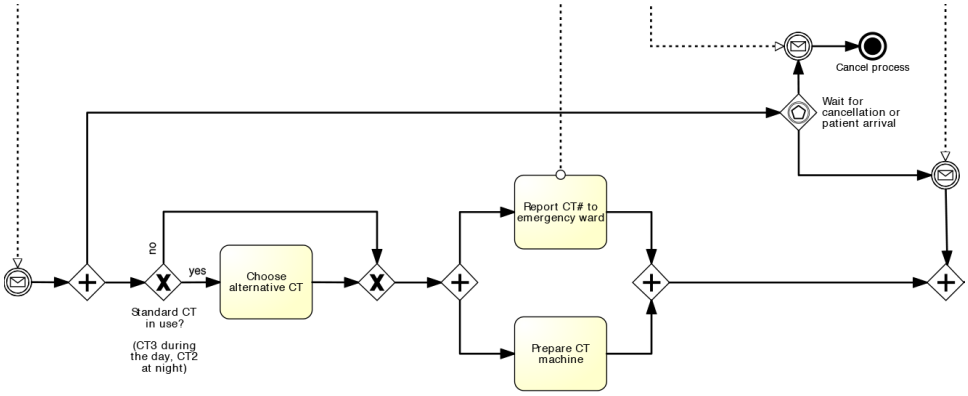

Figure 4.7: Radiology: Process after receiving patient info from the emergency room.

After the emergency room creates the internal and neurological report a decision is made: either it will be determined that the patient needs acute therapy, which ends the emergency rooms involvement in the process and also sends a message to radiology to stand down with the CT preparations which will not be needed anymore or it is determined that the patient needs acute therapy, which results in the sending of the patient to radiology as soon as the transport service and the CT number have arrived at the emergency room (see fig. 4.8). In this case the neurologist will accompany the patient, bringing all available reports and information with him.

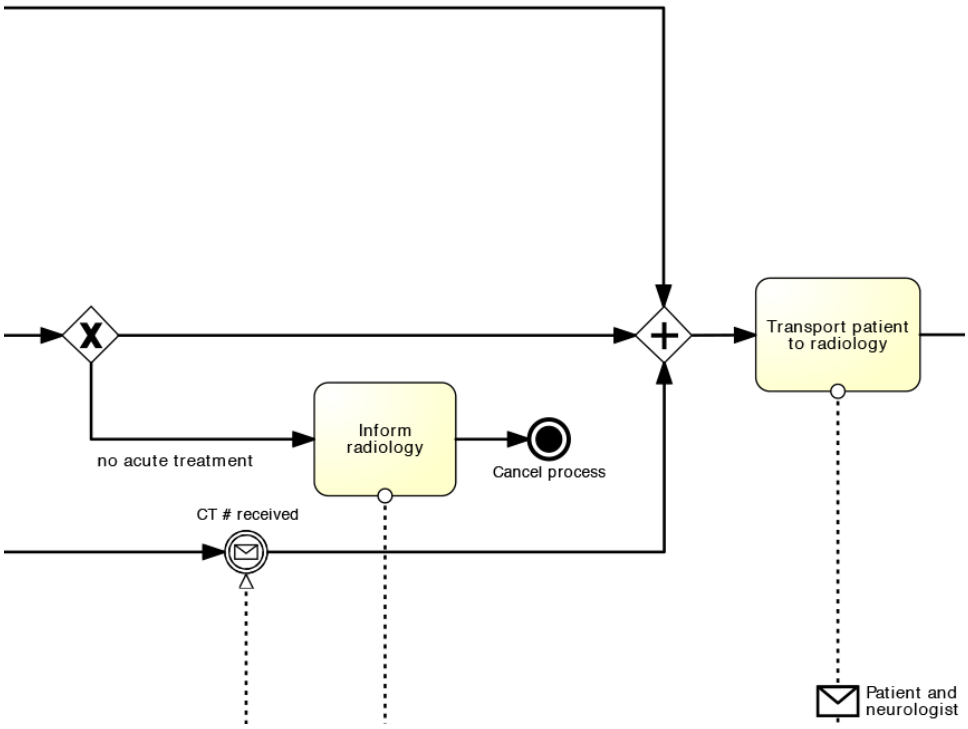

Figure 4.8: Emergency room: Process reports, CT number and transport service are available.

Radiology for its part will wait for the message from the emergency room depending on which it will either terminate its process in case no further therapy is required or receive the patient in order to conduct the CT on the patient. If the patient needs acute treatment, he arrives in the radiology together with the neurologist, where as soon as the CT has been prepared a CT will be conducted on the patient. After the CT a radiological report will be created (see fig. 4.9).

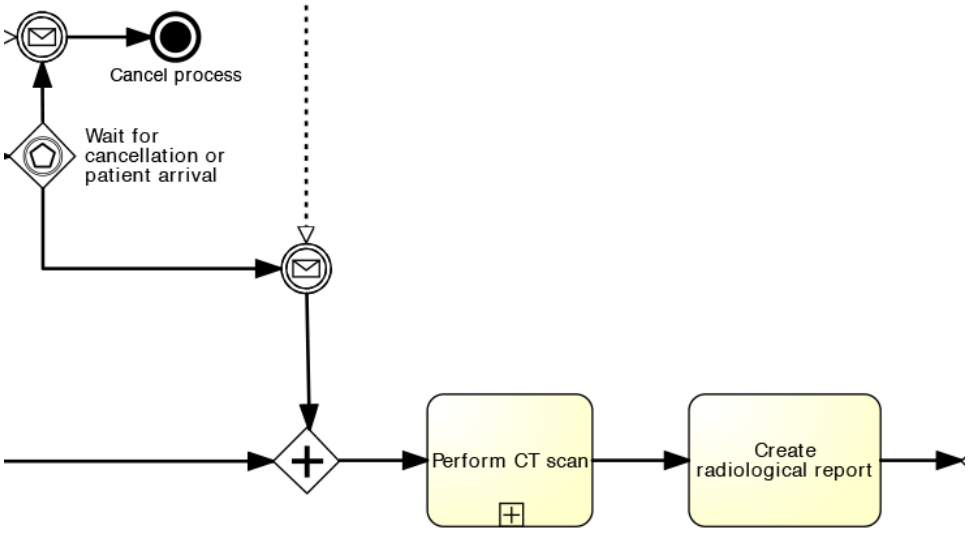

Figure 4.9: Radiology: Process continues according to message of emergency room, depending on whether the patient needs acute therapy or not.

While radiology and emergency room execute the above mentioned process steps, laboratory starts to examine the blood sample it got from the emergency room. As soon as this analysis is done, it will send the information to radiology, which by then should have finished the CT on the patient (see fig. 4.10).

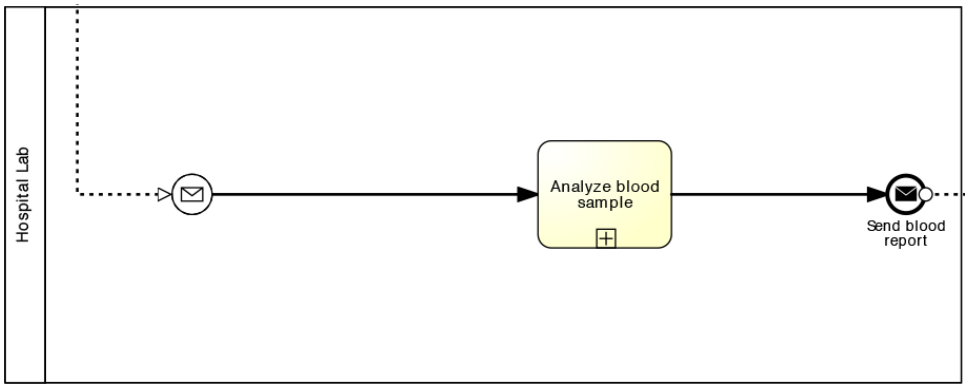

Figure 4.10: Laboratory: Complete process after receiving a blood sample from the emergency room.

After radiology receives the blood test results from the laboratory as well as all other information available on the patient (provided by the neurologist who accompanied the patient to radiology), it will decide if acute therapy from radiology is needed and if so start thrombolysis. After that the patient is sent back to the emergency room. If the radiology judges the patient not to be in need of acute therapy he will be sent back instantly. After releasing the patient to the emergency room the radiology's process terminates (see fig. 4.11).

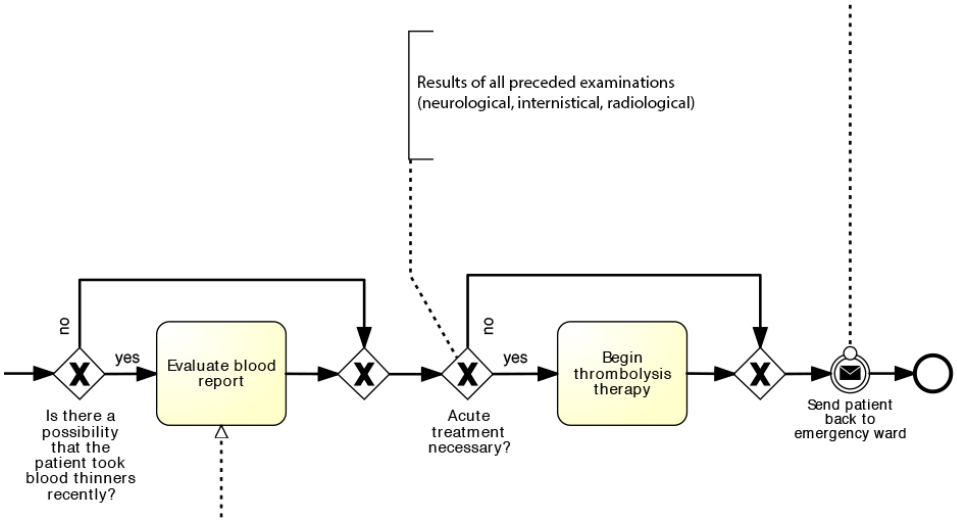

Figure 4.11: Radiology: Process terminates after receiving the blood test results results from the laboratory and then deciding whether the patient needs further acute treatment in the radiology or not.

Finally the emergency room receives the patient from the radiology and will initiate stationary monitoring for him by which the stroke treatment process comes to an end (see fig. 4.12).

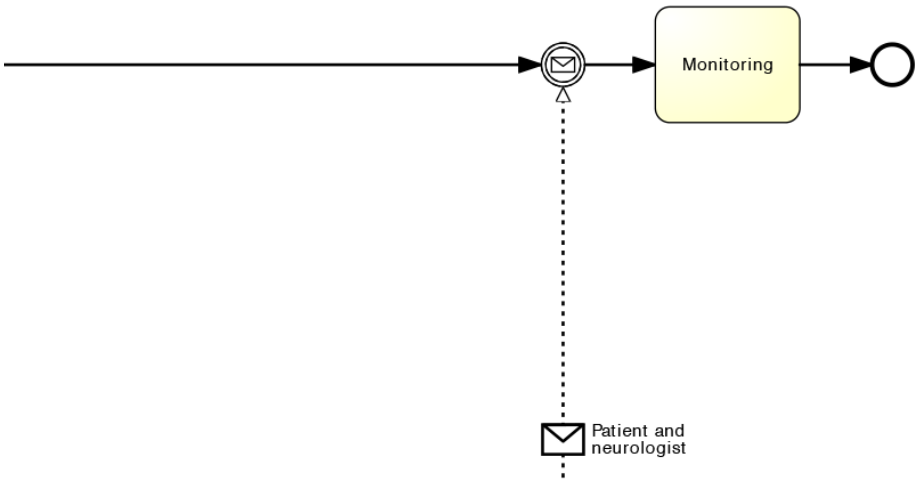

Figure 4.12: Emergency room: Process terminates after patient is returned from the radiology and stationary monitoring has been initiated

Regarded as a dynamic coalition the agents, coalitions and sub-coalitions of this process may be visualized such as in Figure 4.13. At the first level a coalition between five agents may be found: emergency room, neurology, radiology, transport service and laboratory. The first four may be seen as dynamic coalitions themselves: The emergency room utilizes nursing staff and internists for its part of the process, while radiology uses its personnel to conduct the CT and decide if further treatment is needed. Neurology sends a neurologist to the emergency room and transport service assigns a transporter.

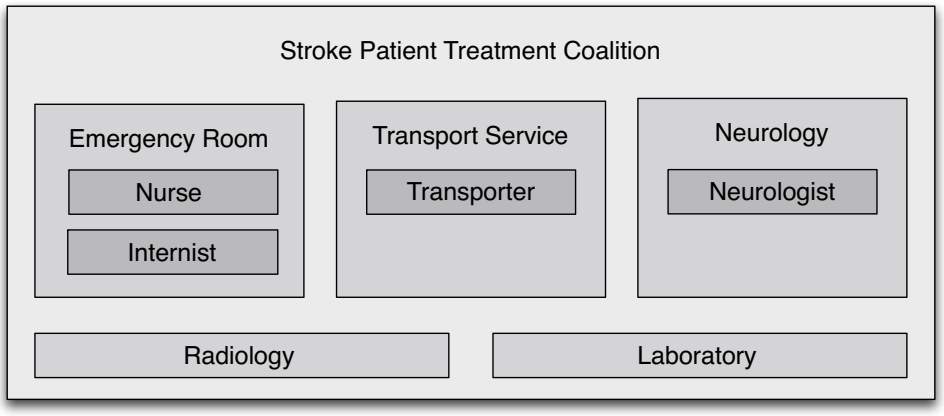

Figure 4.13: Agents, coalitions and sub-coalitions in the stroke patient treatment scenario.

Regarding the events by which the coalition forms and dissolves, the following may be identified:

Coalition forming and dissolving

- At first the stroke treatment coalition consists of the emergency room only which itself is a coalition and carries doctors and nursse and will determine a nurse to conduct the initial examination. Also a patient is joining the ER coalition.
- After the alarm neurology, radiology and transport service join the coalition through the emergency room's imitative. The ER will also determine a doctor internist to conduct the internal examination.
- During the course of the emergency room's internal examination, a neurologist from the neurology coalition joins the stroke treatment coalition as well as the emergency room coalition through the neurology's initiative.
- During the course of the emergency room's internal examination laboratory joins the stroke treatment coalition.
- After the neurologist is sent to the emergency room neurology leaves the coalition and its involvement is ended.
- After the laboratory has analyzed the blood samples and sent the results to radiology it leaves the coalition and its involvement is ended.
- After the transport service sent a transporter to the emergency room, it leaves the coalition and its involvement is ended.
- After the examinations in the emergency room the patient together with the neurologist leave the ER coalition and join the radiology coalition.
- After the radiology has conducted its necessary analyses and possible therapies it leaves the coalition after returning the patient and the neurologist back to the emergency room.
- After the transport of the patient back to the emergency room the transporter leaves the coalition.
- After the patient prepared for stationary monitoring the emergency treatment had come to an end and the coalition terminates with the emergency room leaving.

The following information sharing, creation and updating events are to be identified in this process:

Information creation and sharing

- The neurologist creates a neurological report of the patient in the emergency room.
- The emergency room sends a blood sample to the laboratory.
- The internist creates an internist report of the patient in the emergency room.
- The emergency room shares patient info with radiology.
- Radiology creates a CT number and shares it with the emergency room.
- Emergency room notifies radiology if further treatment is needed and if so sends the patient to radiology
- If the patient is sent to radiology the neurologist will accompany him, bringing along all records and information of the patient. Thus, all patient records and infos are shared with radiology.
- If the patient is sent to radiology, radiology will create a CT and a radiological report.
- Laboratory sends blood work to radiology.
- After treating the patient radiology sends him back to the emergency room.

Up to the current day the information sharing inside of this stroke patient treatment process is mostly handled in an analog way (paper, telephone,etc.). It is the goal of this thesis to provide modeling tools for first step towards a technical solution to that problem. Regarding the process as a privacy-sensitive dynamic coalition, the created model may serve as a foundation for future access controlled IT-infrastructure which is to support the dynamic coalition process and may simplify certain aspects of it (reducing phone calls, sending of papers etc.).

However, the first model will solely capture the existing process, modeling also the analog information interaction in an abstract way, with the basic dynamic coalition model as its foundation. The resulting model may be seen as the equivalent to the first phase of the software-lifecycle mentioned in the first chapter of this thesis: *Requirement capture* of the abstract process according to the *BPMN* specification.

In the subsequent section the *Detailed Design* phase will demonstrate how to integrate access control models into the process, further refining the conditions under which access may take place. These access control mechanisms a critical for the success of a supporting dynamic coalition software infrastructure and should therefore be integrated in the first refinement step of the *detailed design* phase. The applicability of both, *identity-based access control* and *role-based access control* mechanisms are examined and discussed here.

Furthermore, it will then be shown and explained how through simulation the resulting models may be used for validation and documentation.

Note that the *print*-commands in the following code are not required for the *ASM* to execute the program. They are merely a means of visualization of a *ASM* run in a textual manner in form output of the *CoreASM* dialog field such that a first understanding of the *ASM*'s behavior is provided to the user (see output example in figure 4.14). They also sometimes textually describe certain actions of agents, which are mostly internal operations regarded to be less relevant or too specific for these first modeling steps.

4.1.2 Requirement Capture - Basic Dynamic Coalition Model

First the basic dynamic coalition structure and its process are captured in the following *ASM* model. The *ASM* defined in this section is the result of *requirement capture* phase of the software-lifecycle as it basically models the existing process in an abstract way thereby making it the formal basis for an understanding of a software architecture which is to support this process.

The basic functions and rules for this model may be found in section 3.1. Therefore, here only the *ASM* program (init-rule and agent-programs) will be shown here. The presentation will be conducted in a rule-by-rule fashion. In the case of a longer rule, the rule will be explained in parts. For an overview of the whole *ASM* program, check the full *CoreASM*-files on the CD attached to this thesis.

The Init-Rule

The init-rule initializes all agents and variables needed at the beginning of the process. First, agents are created, representing the patient, the emergency room, the neurology, the radiology, the transport service, the laboratory and the stroke treatment coalition. The model supposes that that if further agents are needed as part of a sub-coalition of these pre-defined ones, they will be freshly created by the coalition itself, e.g. the emergency room creating a nurse agent and adding it to the coalition.

The only program to be initialized so far is the emergency room because the patient will always be brought there first and the emergency determines if stroke treatment is necessary and if the other agents should be activated. Thus, initially the

```
CoreASM XACML
ER: Patient Arrived! Contacting nurse.

ER NURSE: Patient apolectiform! Setting Alarm!

RADIOLOGY: Setting up CT number.
RADIOLOGY: Sharing CT number with coalition.
RADIOLOGY: Choosing and preparing CT.

ER INTERNIST: Starting internistic examination!

NEUROLOGY: Contacting Neurologist!
NEUROLOGY: Sending Neurologist to the ER

ER: Sharing Blood Sample for laboratory.

ER: Neurologist arrived.

NEUROLOGIST: Examining Patient. Creating neurological report.

LABORATORY: Creating blood test.

ER: Patient leaving Emergency Room with Neurologist.

TRANSPORTER: Transporting Patient to Radiology.

RADIOLOGY: Patient arrived in Radiology.
RADIOLOGY: Performing CT.
RADIOLOGY: Evaluating Blood Test.
RADIOLOGY: Acute treatment necesarry. Beginning thrombolisis therapy.
RADIOLOGY: Sending Patient back to emergency room.

TRANSPORTER: Transporting Patient to Emergency Room.

ER: Monitoring patient!
```

Figure 4.14: The *CoreASM* output for the *ASM* introduced in section 4.1.2.

stroke treatment coalition consists only of the emergency room. The sub-coalition of the emergency room consists only of the patient.

A number of boolean variables are needed to represent certain attributes of the process or the patient or to serve as variables for the mutual exclusion of the agents steps: *Apolectiform* defines if a patient is actually showing stroke symptoms. Alarm represents the alarm which has to be triggered upon recognizing a stroke. *NurseExam* and *NeurologistExam* are exclusion variables which are true if the corresponding examination took place. *AcuteStatus* defines if the patient is in a state that seems to need acute treatment while *AcuteTreatment* represents the patient's factual need for acute treatment after all examinations have been done. *BloodThinners* defines whether the patient has been taking blood thinners or not. *TransportER* and *TransportRadiology* stand for the assignment to start the transport to radiology or to the emergency room respectively. In this example the patient variables (*Apolectiform*,*AcuteStatus*,*AcuteTreatment* and *BloodThinners*) are set in

such a manner that the process may actually carried out to its very last action supposing that the patient is having a stroke and actually needs the whole stroke treatment process to take place. The exclusion variables (*NurseExam*, *NeurologistExam*, *TransportER*, are initially set to false and will later be modified.

```
rule initRule = {
seqblock
  SetUpEmptyAgent("Patient")
  SetUpEmptyAgent("Neurology")
  SetUpEmptyAgent("Radiology")
  SetUpEmptyAgent("Transport")
  SetUpEmptyAgent("Laboratory")
  SetUpEmptyAgent("StrokeCoalition")
  SetUpEmptyAgent("ER")

  program(CallByName("ER")) := @ERProgram
  AddCoalitionMembers(CallByName("ER"),
       CallByName("StrokeCoalition"))
  AddCoalitionMembers(CallByName("Patient"),CallByName("ER"))

   Apolectiform:= true
   AcuteStatus:= true
   BloodThinners:= true
   AcuteTreatment := true

   Alarm:= false
   NurseExam := false
   NeurologistExam:= false
   TransportRadiology:=false
   TransportER:=false
   Agents(self) := false
endseqblock
}
```

The Emergency Room Agent Program

The rule *ERProgram* is obviously the central part of the stroke treatment process. Because of its length it will be described in parts: the rule consists of two parallel *if*-clauses, each of which will be described separately here and with the second one consisting yet of a number of *if*-clauses which represent the treatment of the patient in case of a triggered stroke alarm.

The program starts out with an *if*-clause which holds *true* when a patient arrives to the emergency room and has not yet been examined by a nurse. The emergency room will then create a new nurse agent and add it to its coalition. The *nurse*-program (see further down) will then be called and the *NurseExam*-variable set to *true* in order to avoid a second execution of this *if*-clause.

```
rule ERProgram = {
  if exists a in CoalitionMembers(CallByName("ER"))
  with a = CallByName("Patient") and not(NurseExam) then
  seqblock
    print "ER: Patient Arrived! Contacting nurse."
    SetUpEmptyAgent("ERNurse")
    AddCoalitionMembers(CallByName("ERNurse"),CallByName("ER"))
    program(CallByName("ERNurse")) := @ERNurseProgram
```

```
    NurseExam:=true
  endseqblock
```

The second *if*-clause reacts to a stroke that might be triggered by the nurse after her examination. If the alarm is triggered, a number of if-clauses run in parallel. The first one checks if a *PatientInfo*-record has been created. If not this means that the process stands at its beginning, which leads to the creation of the *PatientInfo*-record as well as the adding of neurology, radiology, laboratory and transport service to the stroke treatment coalition followed by the activation of their programs. In addition an emergency room internist is created who is needed for the internal examination of the patient in the course of the process. His program also starts right away.

```
if Alarm then {
  if not exists i in AgentInfo(self)
  with i=CallByName("PatientInfo") then
  seqblock
    CreateInfo(self,"PatientInfo")
         ShareInfo(self,CallByName("StrokeCoalition"),
         CallByName("PatientInfo"))
    AddCoalitionMembers(CallByName("Neurology"),
         CallByName("StrokeCoalition"))
    program(CallByName("Neurology")) := @NeurologyProgram
    AddCoalitionMembers(CallByName("Radiology"),
         CallByName("StrokeCoalition"))
    program(CallByName("Radiology")) := @RadiologyProgram
    AddCoalitionMembers(CallByName("Laboratory"),
         CallByName("StrokeCoalition"))
    program(CallByName("Laboratory")) := @LaboratoryProgram
    AddCoalitionMembers(CallByName("Transport"),
         CallByName("StrokeCoalition"))
    program(CallByName("Transport")) := @TransportProgram
    SetUpEmptyAgent("ERInternist")
    AddCoalitionMembers(CallByName("ERInternist"),
         CallByName("ER"))
    program(CallByName("ERInternist")) := @ERInternistProgram
 endseqblock
```

The next *if*-clause inside the alarm *if*-clause checks if a blood sample has been created by the internist agent and if so shares it with the coalition in order to make it accessible for the laboratory, if it has bot been shared already.

```
  if exists i in AgentInfo(self) with i=CallByName("BloodSample")
  and not exists i in AgentInfo(CallByName("StrokeCoalition"))
  with i=CallByName("BloodSample") then
  seqblock
    print "ER: Sharing Blood Sample for laboratory."
    ShareInfo(self,CallByName("StrokeCoalition"),
         CallByName("BloodSample"))
  endseqblock
```

The next clause checks whether the neurologist has arrived and whether he has performed his examination and if not his program will be activated.

```
  if exists a in CoalitionMembers(CallByName("ER"))
  with a = CallByName("Neurologist") and not(NeurologistExam) then
```

```
seqblock
  print "ER: Neurologist arrived."
  program(CallByName("Neurologist")):= @NeurologistProgram
  NeurologistExam:=true
endseqblock
```

The last *if*-clause waits first for the neurological and internal reports to arrive. If the patient is also in the emergency room and the radiological report is still missing it then checks, according to the available reports, if acute treatment seems to be necessary. Indeed this decision is made by a supervising doctor who has to access the neurological and internal reports for his decision taking. However, the decision taking process is omitted here and replaced by the boolean variable *AcuteStatus* which defines whether the process has to go on or terminate.

If the program goes on, it waits for the CT number being shared with the coalition by radiology as well as for the transporter to get to the emergency room. As soon as that happens the emergency room will share its internal report with the coalition and activate the transporter which will take care of the leave of the patient and the neurologist from the emergency room coalition and the join to radiology respectively.

As soon as the radiological treatment report arrives with the patient returning from the treatment in radiology the emergency room tries to access the report and when successful starts to monitor the patient for his stationary stay at the hospital. Thereby the stroke treatment process as a whole is ended correctly.

The last two brackets close the alarm-*if*-clause and the *ERprogram*-rule as a whole respectively.

```
if exists i in AgentInfo(CallByName("StrokeCoalition"))
with i = CallByName("NeurologicalReport")
and exists j in AgentInfo(CallByName("ER"))
with j = CallByName("InternalReport")
and not exists h in AgentInfo(CallByName("StrokeCoalition"))
with h = CallByName("RadiologicalReport")
and exists a in CoalitionMembers(CallByName("ER"))
with a=CallByName("Patient")
and not(TransportRadiology) then {
  seq req1 <- RequestInfo(self, CallByName("StrokeCoalition"),
       CallByName("NeurologicalReport"), read) next
  seq req2 <- RequestInfo(self, CallByName("StrokeCoalition"),
       CallByName("InternalReport"), read) next
  if not (req1 = permit and req2=permit) then
    seq print "ER: Can't access reports!" next shutdown
  else if not(AcuteStatus) then
    seq print "ER: Patient status not acute! Ending process!"
    next shutdown

  else if exists a in CoalitionMembers(CallByName("ER"))
  with a = CallByName("Transporter")
  and exists i in AgentInfo(CallByName("StrokeCoalition"))
  with i=CallByName("CTnumber") then
  seqblock
    ShareInfo(self, CallByName("StrokeCoalition"),
         CallByName("InternalReport"))
    print "ER: Patient leaving Emergency room with Neurologist."
    TransportRadiology:=true
```

```
        program(CallByName("Transporter")):= @TransporterProgram
      endseqblock
    }
    else if exists i in AgentInfo(CallByName("StrokeCoalition"))
    with i = CallByName("RadiologicalReport")  then
      seq req <- RequestInfo(self, CallByName("StrokeCoalition"),
           CallByName("RadiologicalReport"), read) next
      if req = permit then
        seq print "ER: Monitoring patient!" next shutdown
      else seq print "ER: ERROR! Can't access radiological report!"
        next shutdown
  }
}
```

Emergency Room Nurse and Internist programs

After being activated the *ERnurse*-program checks whether the variable *Apolectiform* holds, representing a stroke symptom of the patient. If not, the whole *ASM* is brought to a hold, because in this case stroke treatment would not be necessary. If *Apolectiform* is *true* however, the *Alarm* variable is set to *true* resulting in the triggering of the further process in the emergency room program. After this the *ERnurse*-program terminates as there is no further critical involvement in the following process, at least at the current level of abstraction.

```
rule ERNurseProgram = {
  if Apolectiform then
  seqblock
    print "ER NURSE: Patient apolectiform! Setting Alarm!"
    Alarm:=true
    terminate self
  endseqblock
  else seq print "ER NURSE: Patient not apolectiform!
      Ending stroke treatment process!"
  next shutdown
}
```

The *internist*-program creates blood sample and internal report records which are shared with the emergency room coalition. In order for the emergency room to be able to share it with the top level stroke treatment coalition, the information owner field is changed to the emergency room for both information items. After that the internist agent terminates.

```
rule ERInternistProgram = {
  seqblock
    print "ER INTERNIST: Starting internist examination!"
    CreateInfo(self,"BloodSample")
    ShareInfo(self, CallByName("ER"), CallByName("BloodSample"))
    ChangeInfoOwner(self, CallByName("ER"),
    CallByName("BloodSample"))

    CreateInfo(self,"InternalReport")
    ShareInfo(self, CallByName("ER"), CallByName("InternalReport"))
    ChangeInfoOwner(self, CallByName("ER"),
    CallByName("InternalReport"))
    terminate self
```

```
  endseqblock
}
```

Neurology and Neurologist Programs

As soon as neurology is activated it creates a new *neurologist*-agent which is to join the neurology's coalition. Furthermore, the neurologist will be joined to the stroke treatment coalition because he will have to share information not only with the neurology coalition but also with the top level agents such as emergency room and radiology. Then the neurologist will be sent to the emergency room, represented by the join of the neurologist to the emergency room coalition.

```
rule NeurologyProgram = {
  seqblock
    print "NEUROLOGY: Contacting Neurologist!"
     SetUpEmptyAgent("Neurologist")
     AddCoalitionMembers(CallByName("Neurologist"),
          CallByName("Neurology"))
     AddCoalitionMembers(CallByName("Neurologist"),
          CallByName("StrokeCoalition"))
     print "NEUROLOGY: Sending Neurologist to the ER"
     AddCoalitionMembers(CallByName("Neurologist"),CallByName("ER"))
     terminate self
  endseqblock
}
```

When the *neurologist*-program is activated by the emergency room, the neurologist starts his examination after which he creates a neurological report and shares it with the coalition. From this point on he will not create or contribute any new information items in this model. Therefore, his program will terminate. Note that the involvement of the neurologist agent in the global process does not end: the agent will still join or leave coalitions , which will be triggered by the program of other coalitions (see emergency room or radiology program).

```
rule NeurologistProgram = {
 seqblock
   print "NEUROLOGIST: Examining Patient.
        Creating neurological report."
   CreateInfo(self,"NeurologicalReport")
   ShareInfo(self , CallByName("StrokeCoalition"),
        CallByName("NeurologicalReport"))
   terminate self
  endseqblock
}
```

Transport Service and Transporter Programs

The transport service program upon activation immediately sets up a new agent representing a concrete transporter which will be added to its coalition as well as to the ER's coalition for which the transporter is supposed to carry out services. After the joining of the transporter agent to the ER coalition no more actions are required by the transport service so that it will terminate.

```
rule  TransportProgram = {
  seqblock
```

```
    SetUpEmptyAgent("Transporter")
    AddCoalitionMembers(CallByName("Transporter"),
         CallByName("Transport"))
    AddCoalitionMembers(CallByName("Transporter"),CallByName("ER"))
    terminate self
  endseqblock
}
```

The transporter program reacts to the value of the boolean variables *TransportRadiology* and *TransportER* which stand for the need of a transport to radiology or the ER and are set by the ER or radiology respectively. As soon as *TransportRadiology* is set by the ER program the transporter should start transporting the patient and the neurologist, represented by the leaving of the patient and the neurologist and itself from the ER and their joining with the radiology coalition. After that *TransportRadiology* is set to false. If *TransportER* is set to true by the radiology, the transporter will perform the inverse actions.

```
rule TransporterProgram = {
  if TransportRadiology then
  seqblock
    RemoveCoalitionMember(self, CallByName("ER"))
    AddCoalitionMembers(self,CallByName("Radiology"))
    RemoveCoalitionMember(CallByName("Patient"), CallByName("ER"))
    AddCoalitionMembers(CallByName("Patient"),
         CallByName("Radiology"))
    RemoveCoalitionMember(CallByName("Neurologist"),
         CallByName("ER"))
    AddCoalitionMembers(CallByName("Neurologist"),
         CallByName("Radiology"))
    print "TRANSPORTER: Transporting Patient to Radiology."
    TransportRadiology:=false
  endseqblock

  if TransportER then
  seqblock
    RemoveCoalitionMember(CallByName("Patient"),
         CallByName("Radiology"))
    AddCoalitionMembers(CallByName("Patient"), CallByName("ER"))
    RemoveCoalitionMember(CallByName("Neurologist"),
         CallByName("Radiology"))
    AddCoalitionMembers(CallByName("Neurologist"),
         CallByName("ER"))
    RemoveCoalitionMember(self, CallByName("Radiology"))
    AddCoalitionMembers(self,CallByName("ER"))
    print "TRANSPORTER: Transporting Patient to Emergency Room."
    TransportER:= false
  endseqblock
}
```

Laboratory Program

The laboratory waits for the blood sample to arrive from the emergency room. As soon as it is shared with the stroke treatment coalition, the laboratory will try to access it and if successful create a blood test and share it with the coalition after which it will terminate. In case it can't access the blood sample, i.e. if the laboratory

has not been joined to the stroke treatment coalition, it returns an error, which will result in the whole process getting into a bottleneck resulting in the termination of the *ASM*. Note however, that in the model this case cannot take place because the laboratory is always joined to the coalition at the very beginning of the emergency rooms program. However, this case will become relevant in the next section, when considering the integration of access control mechanisms.

```
rule  LaboratoryProgram = {
  if exists i in AgentInfo(CallByName("StrokeCoalition"))
  with i= CallByName("BloodSample") then
  seqblock
    req <- RequestInfo(self, CallByName("StrokeCoalition"),
         CallByName("BloodSample"), "read")
   if req = permit then
   seqblock
     print "LABORATORY: Creating blood test."
     CreateInfo(self, "BloodTest")
     ShareInfo(self, CallByName("StrokeCoalition"),
     CallByName("BloodTest"))
     terminate self
   endseqblock

   else seq print "LABORATORY: ERROR! Cannot access blood sample"
   next shutdown
endseqblock
}
```

Radiology Program

As the emergency room program before, the radiology program will be shown and explained in parts here. It consists of two if-clauses, with the first one reacting to an alarm, the existence of a patient's information record in the stroke treatment coalition and the non-existence of a CT number for a possible CT for that patient. In case this condition holds, the radiology will try to access the patient info and in case of a successful access create a CT number, share it with the coalition and set the variable *CTprepared* to *true*. Note, that in the *BPMN* model an iteration takes place at this moment, which checks if a certain CT is available and if not finds another one. This particular part of the program is omitted here as it is considered to be an internal operation of radiology, which does not directly affect the work of the stroke treatment coalition, as long as the assumption that one CT always is available holds. If radiology can't access the patient info it will throw an error and as this would constitute a critical error, the *ASM* will terminate.

```
rule RadiologyProgram = {
  if Alarm and exists i in AgentInfo(CallByName("StrokeCoalition"))
  with i=CallByName("PatientInfo")
  and not exists i in AgentInfo(self)
  with i=CallByName("CTnumber") then
  seqblock
    req <- RequestInfo(self, CallByName("StrokeCoalition"),
         CallByName("PatientInfo"), "read")
    if req = permit then
    seqblock
      print "RADIOLOGY: Setting up CT number."
           CreateInfo(self,"CTnumber")
```

```
    print "RADIOLOGY: Sharing CT number with coalition."
    ShareInfo(self, CallByName("StrokeCoalition"),
         CallByName("CTnumber"))
    print "RADIOLOGY: Choosing and preparing CT."
         CTprepared:=true
  endseqblock
  else seq print "RADIOLOGY: ERROR! Cannot access
       patient information"
  next shutdown
endseqblock
```

The next *if*-clause holds as soon as a CT number has been shared with the coalition, a CT has been prepared and the transporter got to radiology carrying patient and neurologist with him. If the condition holds a CT will be performed as a conclusion of which a radiological report will be created and shared with the coalition. In a next step, depending on whether the patient is taking blood thinners, radiology will try to access the laboratory's blood test. If it cannot access the test, the process cannot continue according to specification represented by a termination of the *ASM* in this case.

```
if exists i in AgentInfo(CallByName("StrokeCoalition"))
with i = CallByName("CTnumber") and CTprepared
and exists a in CoalitionMembers(CallByName("Radiology"))
with a = CallByName("Patient")
and exists a in CoalitionMembers(CallByName("Radiology"))
with a = CallByName("Neurologist") then
seqblock
  print "RADIOLOGY: Patient arrived in Radiology."
  print "RADIOLOGY: Performing CT."
  CreateInfo(self, "RadiologicalReport")
  ShareInfo(self, CallByName("StrokeCoalition"),
  CallByName("RadiologicalReport"))
  if BloodThinners then
  seqblock
    req <- RequestInfo(self, CallByName("StrokeCoalition"),
         CallByName("BloodTest"), "read")
    if req = permit then
      print "RADIOLOGY: Evaluating Blood Test."
    else seq print "RADIOLOGY: ERROR Can't access blood test."
    next shutdown
  endseqblock
```

In the next step all information that has been created through the collaborating agents in the stroke coalition will have to be accessed. If the access is granted the rest of the radiology's process continues. If not, as above, the process terminates. In the former case, radiology will check if acute treatment in the radiology is necessary. This decision is determined through the check of the pre-set boolean variable *Acute-Treatment*. If so, the treatment will be performed after which the patient is marked as being ready for stationary treatment and the transporter notified to execute the transport from radiology back to the emergency room. After this the involvement of the radiology is ended, so that the radiology program may terminate.

```
  req1 <- RequestInfo(self, CallByName("StrokeCoalition"),
       CallByName("NeurologicalReport"), "read")
  req2 <- RequestInfo(self, CallByName("StrokeCoalition"),
```

```
        CallByName("InternalReport"), "read")
    req3 <- RequestInfo(self, CallByName("StrokeCoalition"),
        CallByName("RadiologicalReport"), "read")
    if req1 = permit
    and req2 = permit
    and req3 = permit then
      if AcuteTreatment then
      seqblock
        print "RADIOLOGY: Acute treatment necesarry.
             Beginning thrombolisis therapy."
        print "RADIOLOGY: Sending Patient back to emergency room."
        TransportER:= true
        terminate self
      endseqblock
      else
      seqblock
        print "RADIOLOGY: Acute treatment not necessary."
        print "RADIOLOGY: Sending Patient back to emergency room."
        TransportER:= true
        terminate self
      endseqblock
    else seq print "RADIOLOGY: ERROR! Can't access reports."
    next shutdown
endseqblock
}
```

4.1.3 Detailed Design - DCs with IBAC/RBAC Model

Whereas during *requirement capture*, the first phase of the software engineering life cycle (see section 1.1.3), the basic dynamic coalition process has been recorded and modeled, in the second phase called *detailed design* the refinement of the model takes place, bringing it closer to an actual implementation specification. Access control mechanisms for privacy-enforcement are critical for the success of a project in the medical field. Therefore, it is critical that integrating these access control mechanisms and adapting the model to the privacy requirements is considered as early as possible. Thus, in this section it is considered as the first refinement step for the *detailed design* phase. Integrating access control mechanisms details the process specification, brings it closer to a real-life implementation and also allows for the testing and validation of access control policies against the process requirements.

In order to integrate access control into the basic dynamic coalition model presented in the previous section, the program has to be adapted to the functions and rules from the *dimension access control* introduced in section 3. First, it has to be decided whether the scenario at hand is to be seen as a dynamic coalition with low or with medium membership dynamics. In order to illustrate this decision process, this section will start out with the assumption that membership dynamics in the dynamic coalitions at hand are low: because all the actors are known in advance (because they are members of the same institutions with contact information, IDs, etc.) this decision seems reasonable.

Access control rules are applied as soon as an access takes places. In the above depicted model these accesses are identified through calls of the rule *Request()*. Namely the accesses which are to be controlled are:

Accesses to be controlled

- The laboratory trying to access the blood sample in order to conduct the blood test.
- The radiology trying to access the patient info in order to prepare CT and create CT number.
- The radiology trying to access the blood test from the laboratory in order to determine whether blood thinners have been used.
- The emergency room trying to access the internal and neurological reports in order to determine if the patient is in an acute status.
- The radiology trying to access all created reports (neurological, internal and radiological report) for the decision whether the patient needs thrombolisis or not.
- The emergency room trying to access the radiological report for the last monitoring phase.

If these accesses are not explicitly granted the process will not function according to specification and the *ASM* program will run into dead ends which do not allow all its parts to be executed.

Detailed design does of course not end with the first policy definition. Further refinement steps would be required in order to reduce the gap between the abstract model and a final software implementation. These steps would include further refinement of the coalition's and subcoalition's processes and interactions as well as policy refinement.

Enforcing Access Control through IBAC

In order to allow these and only these accesses the corresponding information owner will have to create access control rules which allow these particular access rights. For example the emergency room being the new owner of the blood sample which was initially created by an internist will have to create an access control rule which allows the laboratory to access the needed information on a read basis. Thus, the following changes have to be made to the part of the emergency rooms code which deal with the blood sample:

```
if exists i in AgentInfo(self) with i=CallByName("BloodSample")
and not exists i in AgentInfo(CallByName("StrokeCoalition"))
with i=CallByName("BloodSample") then
seqblock
  print "ER: Sharing Blood Sample for laboratory."
  ShareInfo(self,CallByName("StrokeCoalition"),
  CallByName("BloodSample"))
  SetUpACRule(self,{CallByName("Laboratory")},
       {CallByName("BloodSample")},{read},permit)
endseqblock
```

The newly created access control rule will be shared with all coalitions with which the corresponding blood sample has been shared with, thereby enforcing that only the Laboratory may access it. All the other parts of the program have to be adapted correspondingly in order to allow the above mentioned accesses. For an overview of the whole *ASM* program, check the full *CoreASM*-files on the CD attached to this thesis.

Discussion: IBAC vs. RBAC

After creating *identity-based access control* rules and extending the model so that they may be enforced the model was presented to medical personnel. From these meetings it became clear that the *identity-based access control* rules impose restrictions on the scenario which are too tight: agents like "transport er" or "neurologist" do not actually mean unique individuals, but in the sense of the model represent the role "transporter" or role "neurologist" respectively. This concept is not supported by the *identity-based access control* approach. If it is supposed for example, that the neurology has a number of neurologists agents to choose from, it would become cumbersome to define access control rules for every single one of them. On the previous pages this particular scenario has not been modeled but supposed the following changes to the neurology program:

```
rule NeurologyProgram = {
  seqblock
    print "NEUROLOGY: Contacting Neurologist!"
    SetUpEmptyAgent("NeurologistA")
    SetUpEmptyAgent("NeurologistB")
    SetUpEmptyAgent("NeurologistC")
    choose N in {CallByName("NeurologistA"),
    CallByName("NeurologistB"), CallByName("NeurologistC")} do {
      AddCoalitionMembers(N, CallByName("Neurology"))
      AddCoalitionMembers(N, CallByName("StrokeCoalition"))
      print "NEUROLOGY: Sending Neurologist to the ER"
      AddCoalitionMembers(N, CallByName("ER"))
      terminate self
     }
  endseqblock
}
```

The *neurology*-program will now non-deterministically choose one of the available *neurologist*-agents making it impossible for the other agents to predict which neurologist is actually going to interact with them. Consider again the following lines from the emergency rooms program:

```
    if exists a in CoalitionMembers(CallByName("ER"))
    with a = CallByName("Neurologist") and not(NeurologistExam) then
    seqblock
      print "ER: Neurologist arrived."
      program(CallByName("Neurologist")):= @NeurologistProgram
      NeurologistExam:=true
    endseqblock
```

If roles are not considered here, these lines would have to be complicatedly adapted to integrate all possible neurology agents into the conditions. The same is true for the corresponding access control rules. In order to make sure, that the process can terminate correctly, every possible access scenario would have to be modeled, which may be straightforward considering a small number of agents but quickly becomes confusing as the number of agents rises above a certain level.

Therefore it seems logical to consider the dynamic coalition scenario at hand as one medium membership dynamics, in which a number of actors are well known at the beginning, but the actual agents which collaborate in the process may be non-deterministic.

Enforcing Access Control through *RBAC*

In order to integrate *role-based access control* into the presented process first the *Init-Rule* has to be extended with the creation of initial roles which are to be considered during the course of the process. The roles which are critical to the access of information initially are: *Radiology, ER, Neurology.* The agents will be assigned to their corresponding roles. Roles for *Laboratory* and *Neurologist* will be added during the course of the process by the hands of the emergency room and neurology as they are responsible for the creation of the agents *Laboratory* and *Neurology* respectively. Therefore in the *Init-Rule* neurology and emergency room agents gain role administration rights. Thus, the following lines have to be added to the *Init-Rule* presented earlier in this section:

```
RoleAdmin:={self, CallByName("ER"),CallByName("Neurology")}

SetUpRole(self,"RadiologyRole")
SetUpRole(self,"ERRole")
SetUpRole(self,"NeurologyRole")
SetUpRole(self,"NeurologistRole")
SetUpRole(self,"LaboratoryRole")

AssignRole(self,CallByName("Radiology"),CallByName("RadiologyRole"))
AssignRole(self,CallByName("Neurology"),CallByName("NeurologyRole"))
AssignRole(self,CallByName("ER"),CallByName("ERRole"))
```

In order to grant access control to a certain role, an access permission has to be set up and assigned to the considered role. It has also to be made sure that the agent which is to receive the access control rights is associated with that particular role. Consider the following lines of the emergency room program which are executed right after the detection of the alarm. After the creation of the patient information record, a read-permission is set after which the permission is assigned to the *radiology*-role. The permissions will be shared with any coalition that the corresponding information is shared with. Since the role has already been assigned to the *radiology*-agent, the radiology will now gain access to the patient info when requested.

```
if Alarm then {
  if not exists i in AgentInfo(self)
  with i=CallByName("PatientInfo") then
  seqblock
    CreateInfo(self,"PatientInfo")
    SetUpPermission(self,CallByName("PatientInfo"),read)
    AssignPermission(self,CallByName("RadiologyRole"),
         PermissionID(CallByName("PatientInfo"),read))
    ShareInfo(self,CallByName("StrokeCoalition"),
         CallByName("PatientInfo"))
```

Furthermore, it is now possible to deal with the interaction of agents in a more dynamic way. For example the problematic case in the previous section, concerning the non-deterministic choosing of a neurologist out of a number of neurologists may now be solved through role assignment. Note that neurology as a role administrator is creating a new role assignment:

```
rule NeurologyProgram = {
seqblock
```

```
   print "NEUROLOGY: Contacting Neurologist!"
   SetUpEmptyAgent("NeurologistA")
   SetUpEmptyAgent("NeurologistB")
   SetUpEmptyAgent("NeurologistC")
   SetUpRole(self, "NeurologistRole")
   choose n in {CallByName("NeurologistA"),
   CallByName("NeurologistB"), CallByName("NeurologistC")} do
   {
      AssignRole(self,n,CallByName("NeurologistRole"))
      AddCoalitionMembers(n, CallByName("Neurology"))
      AddCoalitionMembers(n, CallByName("StrokeCoalition"))
      print "NEUROLOGY: Sending Neurologist to the ER"
      AddCoalitionMembers(n, CallByName("ER"))
  }
  terminate self
endseqblock
}
```

The emergency room now does not have to address a specific neurologist but will indeed check for an agent with the role neurologist and choose that particular agent to be assigned to the neurologist's program, allowing for any of the three agents *NeurologistA*, *NeurologistB* or *NeurologistA* to take part in the process:

```
    if exists a in CoalitionMembers(CallByName("ER"))
    with  UserRoles(a) intersect {CallByName("NeurologistRole")}
    = {CallByName("NeurologistRole")}
    and not(NeurologistExam) then
    seqblock
      print "ER: Neurologist arrived."
      choose a in CoalitionMembers(CallByName("ER"))
      with exists r in  UserRoles(a)
      with r = CallByName("NeurologistRole") do
        program(a):= @NeurologistProgram
      NeurologistExam:=true
    endseqblock
```

All the other parts of the program have to be adapted correspondingly in order to allow the above mentioned accesses. For an overview of the whole *ASM* program, check the full *CoreASM*-files on the CD attached to this thesis.

4.2 Research Data Exchange at the Newborn Hearing Screening Berlin

A second case study taken from the pool of case studies of the *SOAMED Research Training Group* deals with the development of a research data exchange platform for the *Newborn Hearing Screening Center* situated at *Charité Virchow Berlin*. It will be shown that the collaborations in this scenario may be regarded as dynamic coalitions with high membership dynamics. Therefore, the possibility of modeling the access control requirements with *attribute-based access control* (*ABAC*) policies for dynamic coalitions with high membership dynamics will be investigated. This approach envisions a fully automated management of access rights. However, from meetings with the medical personnel associated with this scenario it became evident, that a semi-automated management may be necessary in which the software is

responsible for the risk assessment (i.e. trust evaluation) of a requester and the final decision to grant access has to be made by a responsible individual. This kind of risk assessment is taken into account by *TBAC* because the notion of *trust* builds upon the terms of *experience* and *risk*. Therefore, the following case study will make use of both models for dynamic coalitions with *ABAC* and *TBAC* respectively.

Note that the scenario definition takes place on a very abstract level in form of a *ground model* which identifies the main components and activities of the coalition, and allows for a later refinement and detailed modeling of the vast amount of details that scenarios like these usually provide.

4.2.1 Informal Scenario Description

An informal description of the research data exchange platform scenario which is to be implemented at the *Newborn Hearing Screening Center Berlin* could be as follows:

> During the examinations of babies and children in the course of the *Newborn Hearing Screening* at the *Charité Berlin* a great amount of patient data is stored at the screening center. Researchers from various disciplines frequently request access to an anonymized set of this data as input for further research. To this day the access decision for this data is handled manually by the screening center's administrator, including an assessment of the requester through telephone calls and science impact factor. A research data exchange platform is to support the administrator either automatically or semi-automatically in form of suggestions as to whether access should be allowed or not. The administrator would then define a set of attributes or trust values which are to be met by the requester in order to gain access and the access response/recommendation is evaluated according to these parameters of the requester.

Regarded as dynamic coalitions, the agents and the collaborations may be visualized such as in Figure 4.15. The central agent of the coalition is the *Newborn Hearing Screening Center (*NHSC*)* which may form coalitions with researchers or group of researchers who want to access data and also with data contributors being doctors who perform screenings on newborns and kids. Researchers may also be contributors themselves. Each agent joins a coalition with the screening center. The coalition that is formed is highly dynamic in the sense, that the complete set of agents that will collaborate is not previously known, and in the sense that agents often join the coalition for a single interaction only and leave again afterwards.

A first set of parameters which are to be included into the access control decision evaluation and are to be provided by the requester is defined as follows:

- Impact factor of the requester in the scientific area in question
- Rating of former research data interaction
- Rating after a direct talk with the requester, if available

Regarding the temporal relations in which coalitions form, no specific order may be identified: Coalitions form and dissolve according to the requester's need and to the screening centers decision.

The following information sharing, creation and updating events are to be identified in this process:

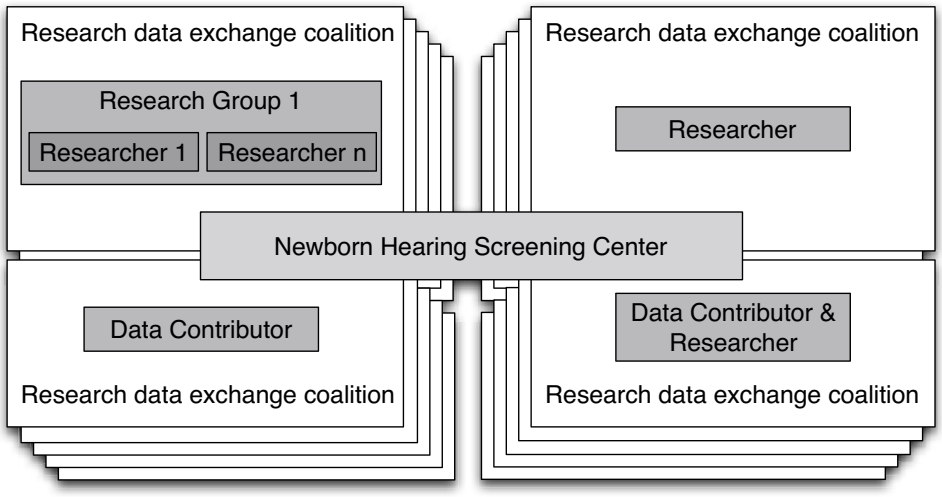

Figure 4.15: Agent and coalitions in the research data exchange scenario.

Information creation and sharing

1. The screening center regularly produces new *patient data*. Subsets or permutations of this data may be classified: The full screening data with all examination curves and personal information has the highest confidentiality. An subset of this data which only contains the information that an examination has been taken place and in which state is of less confidentiality. Summary data which provides statistical analysis over the whole set of data, is completely anonymized and contains no examination results and therefore has the least confidentiality.
2. The screening center records for each agent a rating of former interaction, if it has already taken place.
3. The screening center records a rating after a direct talk with the requester.
4. Requester agents may publish scientific works, affecting their scientific impact factor.
5. Requester may be doctors who contribute screening data to the *NHSC*.

Different than in the first case study of this thesis the one at hand does not deal with an already established process in the medical field, but is rather an initial design of a platform to be. It may therefore be seen as a first step towards an actual implementation of a software which is meant to create dynamic coalition process. First a *requirement capture* defines the process and the agent actions which are to be supported. In the *detailed design* phase the privacy-sensitive information items are then protected with the access control mechanism of choice ($ABAC$ or $TBAC$). Then, through simulation it will be shown that the created access control policies work according to plan and finally it will be argued how the resulting model may serve as a documentation both for the initial specification of the exchange platform as well as for an ongoing documentation model which may be refined and adapted throughout the actual implementation phase.

Note that the *print*-commands in the following code are not required for the *ASM* to execute the program. They are merely a means of visualization of an *ASM* run in a textual manner in form output of the *CoreASM* dialog field such that a first understanding of the *ASM*'s behavior is provided to the user (see output example in figure 4.16). They also sometimes textually describe certain actions of agents, which are mostly internal operations regarded to be less relevant or too specific for

these first modeling steps.

```
CoreASM NHCDC
NHC Initializing!

RESEARCHER: R3 Trying to access Summary Data!

NHC: ACCESS DENIED FOR REQUESTER R3!
RESEARCHER: R3 is joining the Coalition!

NHC: Calling researcher R3!

RESEARCHER: R1 Trying to access examination outcome!

NHC: ACCESS DENIED FOR REQUESTER R1!
RESEARCHER: R1 is joining the Coalition!

NHC: Calling researcher R1!

RESEARCHER: R3 Trying to access examination outcome!

NHC: ACCESS GRANTED FOR REQUESTER R3!


RESEARCHER: R1 Trying to access examination outcome!

NHC: ACCESS GRANTED FOR REQUESTER R1!


RESEARCHER: R2 is joining the Coalition!

RESEARCHER: R2 Trying to access examination data!

NHC: ACCESS DENIED FOR REQUESTER R2!
RESEARCHER: R2 Trying to access Summary Data!

NHC: ACCESS GRANTED FOR REQUESTER R2!


RESEARCHER: R2 Contributing data for screening center!
NHC: Calling researcher R2!
```

Figure 4.16: An example *CoreASM*-output for the *ASM* introduced in section 4.2.2.

4.2.2 Requirement Capture - Basic Dynamic Coalition Model

First the basic dynamic coalition structure and its process are captured in the following *ASM* model. The *ASM* defined in this section is the result of *requirement capture* phase of the software-lifecycle because it basically models the existing process in an abstract way thereby making it the formal basis for an understanding of a software architecture which is to support this process.

The basic functions and rules for this model are identical with the ones found in section 3.1 with two exceptions: First the rule *AddCoalitionMember()* is extended by a function `Contacted: Agents * Agents -> BOOLEAN` which records whether an agent has already been contacted for a direct talk over the phone:

```
rule AddCoalitionMembers(ag, co) ={
```

```
  add ag to CoalitionMembers(co)
  Contacted(co,ag):=false
}
```

Furthermore, in order to integrate a certain non-determinism concerning the requested resources the following helper rule returns a random number between 0 and 1:

```
rule  CreateRandom = {
  choose x in [1 .. 100] do
  result:= x / 100
}
```

All the other rules and functions of the following *ASM* program coincide with the basic dynamic coalition model in section 3.1. Therefore in the following, only the *ASM* program (init-rule and agent-programs) will be shown here. The presentation will be conducted in a rule-by-rule fashion. In the case of a longer rule, the rule will be explained in parts. For an overview of the whole *ASM* program, check the full *CoreASM*-files on the CD attached to this thesis.

The Init-Rule

The init-rule initializes all agents and variables needed at the beginning of the process. First, agents are created, representing the *newborn hearing screening center* and the requesting agents. The number of requesting agents may vary, according to the modelers wishes. In this example three agents with an identical program are initialized.

```
rule initRule = {
  seqblock
    print "Newborn Hearing Center (NHSC) Initializing!"
    SetUpEmptyAgent("NHSC")
    program(CallByName("NHSC")) := @NHSCProgram

    SetUpEmptyAgent("R1")
    program(CallByName("R1")) := @R1Program
    SetUpEmptyAgent("R2")
    program(CallByName("R2")) := @R1Program
    SetUpEmptyAgent("R3")
    program(CallByName("R3")) := @R1Program
    /*create as many agents as needed*/
    Agents(self) := false
  endseqblock
}
```

The Newborn Hearing Screening Center Program

The rule *NHSCProgram* represents the process of the *NHSC* in the course of the research data exchange platform. It is the source of the created information which is supposed to be shared. For a first modeling attempt, the program abstractly represents the huge amount of data in three classification information items, each one representing high, medium and low confidential data respectively. This data is created once at the beginning at the *NHSC*-program. Naturally, the information storage of the newborn hearing center grows over time, and both the center itself, as well as other doctors, are contributing to that growth. However, for the purpose of

this model, the explicit representation of the single data items is not critical. Hence, the actual content of these classification items is omitted here. Note that the created information is not shared with any coalition: In this model the *NHSC*-agent itself represents the coalition, with which other agents may collaborate.

```
rule NHSCProgram = {
  if AgentInfo(self) = {} then
  seqblock
    CreateInfo(self,"HighConfidentialData")
    CreateInfo(self,"MediumConfidentialData")
    CreateInfo(self,"LowConfidentialData")
  endseqblock
```

In the second part of the *NHSC*'s program it tries to contact arriving requesters by telephone represented by the following lines. The *choose*-rule randomly selects one of the agents present in the coalition and sets the *Contacted()*-function for the *NHSC*-coalition and the requester in question to *true*:

```
  choose a in CoalitionMembers(self)
  with not(Contacted(self,a)) do
  seqblock
    print "NHSC: Calling researcher " +Name(a)+ "!"
         Contacted(self,a):=true
  endseqblock
}
```

Requester Programs

In this example three requester agents have been initialized. However their programs are identical, providing a number of non-deterministic options of actions to choose from as well as some conditional actions bound to a certain probability. Naturally the programs of each agent could be modeled differently. This identical representation was chosen, because it represents all the action an agent could take in this scenario, without any restrictions to the order in which they appear. This generic approach also allows for an easier extension of the simulation for a scenario which a much greater number of agents because by simple assignment to this single program a new agent may be initialized.

The first part of the program checks if the reqeuester has already been added to the coalition of the newborn hearing center and if not takes this action, representing the agents desire to collaborate with the *NHSC*:

```
rule RequesterProgram = {
  if not exists a in CoalitionMembers(CallByName("NHSC"))
  with a = self then
  seqblock
    print "RESEARCHER: " +Name(self)+ " is joining the Coalition!"
         AddCoalitionMembers(self, CallByName("NHSC"))
  endseqblock
```

Next, the requester program non-deterministically chooses from requesting access to one of the information types. In this basic scenario the access will always be granted as soon as the requester has joined the coalition and denied if he has not yet joined.

```
  choose action in "a","b","c" do
```

```
seqblock
  if action = "a" then
    seq print "RESEARCHER: " +Name(self)+ " Trying to access
         summary data!"
    next e <- RequestInfo(self, CallByName("NHSC"),
         CallByName("LowConfidentialData"), read)

  if action = "b" then
    seq print "RESEARCHER: " +Name(self)+ " Trying to access
         examination outcome!"
    next e <- RequestInfo(self, CallByName("NHSC"),
         CallByName("MediumConfidentialData"), read)

  if action = "c" then
    seq print "RESEARCHER: " +Name(self)+ " Trying to access
         Summary Data!"
    next e <- RequestInfo(self, CallByName("NHSC"),
         CallByName("HighConfidentialData"), read)
endseqblock
```

Furthermore *requester*-agents may contribute screening data to the newborn hearing screening center. Scientific publications are also relevant for the later evaluation of access control rights. Therefore, requesters may also publish scientific publications. In this basic model these actions are dummy operations which simply print an output message and do not update the state of this machine at all.[2] A third possible action is the researchers decision to leave the coalition. All three actions are associated with a low probability reflecting the fact, that they do not happen very frequently (for the contribution and the leave in 10% and for the publication in 20% of the cases are assumed). The exact probability may be chosen by the programmer and may indeed be different for every simulation run. Thus, the last lines of the requester program are:

```
seqblock
  r2 <- CreateRandom
  if r2 > 0.9 then
    print "RESEARCHER: " +Name(self)+ " Contributing data for
         screening center!"
endseqblock

seqblock
  r2 <- CreateRandom
  if r2 > 0.8 then
 print "RESEARCHER: " +Name(self)+  " Publishing scientific
      works!"
endseqblock
```

[2]This is motivated as follows: First, the screening data in this ground model is not explicitly modeled with all its information items. Therefore, it is not necessary to model the adding of single information items to the screening data either. If the fact that a contribution took place is of importance this event could be recorded through a boolean variable, as will be shown in the next section. Second, for the final implementation of this exchange platform it is envisioned, that it will make use of several online portals in order to determine the impact factor of a scientist. Therefore, the information item of the publication itself is of little relevance for this implementation. However both actions are still shown here as a wild card for the later needed update of the state in order to change attributes or trust values associated with the contribution of publication respectively.

```
  seqblock
    r3 <- CreateRandom
    if r3 > 0.9 then
    seqblock
   print "RESEARCHER: " +Name(self)+  " exiting coalition!"
      RemoveCoalitionMember(self, CallByName("NHSC"))
      terminate self
    endseqblock
  endseqblock
}
```

4.2.3 Detailed Design - DCs with ABAC/TBAC Model

Whereas during *requirement capture*, the first phase of the software engineering life cycle (see section 1.1.3) the basic dynamic coalition process has been recorded and modeled, in the second phase called *detailed design* the refinement of the model takes place, bringing it closer to an actual implementation specification. Access control mechanisms for privacy-enforcement are critical for the success of a project in the medical field. Therefore, it is critical that integrating these access control mechanisms and adapting the model to the privacy requirements is considered as early as possible. Thus, in this section it is considered as the first refinement step for the *detailed design* phase. Integrating access control mechanisms details the process specification, brings it closer to a real-life implementation and also allows for the testing and validation of access control policies against the process requirements.

Therefore in this second step, the model will be adapted and extended in order to make way for access control. Which access control mechanism is actually needed is determined by the degree of the coalition's membership dynamics. Since the possible requesters which join a coalition are not known in advance, the scenario constitutes a dynamic coalition with high membership dynamics. In this example it is shown how both *attribute-based access control* and *trust-based access control* may be applied and both bring their own advantages. However, it will be argued, that because the resulting real-life implementation is not necessarily supposed to work fully automatic but is rather envisioned to serve as a risk assessment and recommendation tool, the integration of *trust* seems to be more adequate.

In order to integrate access control into the basic dynamic coalition model presented in the previous section, the program has to be adapted to the functions and rules from the *dimension access control* introduced in section 3.

Access control rules are applied as soon as an access takes places. In the above depicted model these accesses are identified through calls of the rule *request*. Namely the accesses to be controlled are:

Accesses to be controlled

- A requesters trying to access information with high confidentiality, namely the full screening information, including personal information, examination curves etc.
- A requesters trying to access information with medium confidentiality, namely the information that and in which state an examination took place.
- A requesters trying to access information with low confidentiality, namely summary data, providing only overall numbers and statistics in a completely anonymized manner.

The envisioned research data exchange platform is supposed to manage these accesses. Therefore, in the upcoming policy definition these accesses have to be recorded in access control policies.

Detailed design does not course not end with the first policy definition. Further refinement steps would be required in order to reduce the gap between the abstract model and a final software implementation. These steps would include further refinement of the coalition's and subcoalition's processes and interactions as well as policy refinement.

Enforcing Access Control through ABAC

In order to enforce access control according to ABAC, the *ASM* program introduced in the previous section has to be modified in the following way:

The *NHSC* agent has to create access control rules. The conditions for these rules are specified through the set of attached conditions. Hence, a number of conditions have to be created to allow the access only under these conditions. Consider the following lines of the *NHSCprogram* after the first creation of the information. *c1* specifies the condition that the targeted resource is the information item *LowConfidentialData*. *c2* states that an ImpactFactor greater than 1 has to be present in the requests attributes, c3 stating that the rating after a telephone conversation with the agent should be higher than 1 and so on. The four created access control rules at the end of the following lines then state that a) a requester with impact factor and telephone rating greater than one may access low confidential data, b) a data contributor may access low confidential data, c) a requester with impact factor 5 or d) a requester with a telephone rating of 5 may access low confidential data.

```
rule NHSCProgram = {
  if AgentInfo(self) = {} then
  seqblock
    CreateInfo(self,"HighConfidentialData")
    CreateInfo(self,"MediumConfidentialData")
    CreateInfo(self,"LowConfidentialData")
    c1 <- CreateRuleCondition("Resource",
         CallByName("LowConfidentialData"),"=")
    c2 <- CreateRuleCondition("ImpactFactor",1,">")
    c3 <- CreateRuleCondition("TelRating",1,">")
    c4 <- CreateRuleCondition("Contribution",true,"=")
    c5 <- CreateRuleCondition("ImpactFactor",5,"=")
    c6 <- CreateRuleCondition("TelRating",5,"=")

    SetUpACRule(self, c1, c2, c3, permit)
    SetUpACRule(self, c1,c4,permit)
    SetUpACRule(self, c1,c5,permit)
    SetUpACRule(self, c1,c6,permit)
```

Similar rules may be created for the other two information classes. In the example at hand the only difference for the access control rules concerning data of medium confidentiality is a higher requirement for condition c2 and c3, namely the impact factor, as well as the telephone rating, has to be at least 3. High confidential data is not to be accessed by anyone but by the information owner, i.e. the *NHSC* agent, therefore no rules will be created for it:

```
    c1 <- CreateRuleCondition("Resource",
         CallByName("MediumConfidentialData"),"=")
    c2 <- CreateRuleCondition("ImpactFactor",3,">")
    c3 <- CreateRuleCondition("TelRating",3,">")
    c4 <- CreateRuleCondition("Contribution",true,"=")
    c5 <- CreateRuleCondition("ImpactFactor",5,"=")
```

```
    c6 <- CreateRuleCondition("TelRating",5,"=")

    SetUpACRule(self, c1, c2, c3, permit)
    SetUpACRule(self, c1,c4,permit)
    SetUpACRule(self, c1,c5,permit)
    SetUpACRule(self, c1,c6,permit)
  endseqblock
```

Furthermore, when contacting a requester by telephone, the *NHSC* will assign a rating attribute for the requester. For the purpose of this model, the associated will be a random number between 1 and 5:

```
  choose a in CoalitionMembers(self)
  with not(Contacted(self,a)) do
  seqblock
    print "NHSC: Calling RESEARCHER " +Name(a)+ "!"
         r1 <- CreateRandom
    rating:= round(r1*5)
    print "NHSC: Setting telephone rating to "+rating+" after
         telephone call!"
    SetSubjectAttribute("TelRating",rating,a)
    Contacted(self,a):=true
  endseqblock
}
```

The requester's program has to be adapted such a way, that the two dummy operations for a scientific publication and for a screening data contribution have an effect on the set of attributes the requester holds. The following lines state that after a scientific publication a random number will be created which will be assigned to a new subject attribute of the requester in case he does not carry an attribute of this type yet. In the other case of him already having an impact factor attribute the attributes value will only be updated if it is bigger than the current value: [3]

```
  seqblock
    r1 <- CreateRandom
    if r1 > 0.8 then
    seqblock
      print "RESEARCHER " +Name(self)+ " Publishing
           scientific works!"
      r <- CreateRandom
      IFrating:= round(r*5)
      if not(exists attr in SubjectAttributes(self)
      with AttributeType(attr)="ImpactFactor") then
        seq print "RESEARCHER " +Name(self)+ " Setting
        impact factor to "+IFrating+" after publication!"
        next SetSubjectAttribute("ImpactFactor",IFrating, self)

      else choose attr in SubjectAttributes(self)
      with AttributeType(attr)="ImpactFactor" do
        if IFrating > AttributeValue(attr) then
    seq print "RESEARCHER " +Name(self)+ " Raising
         impact factor to "+IFrating+" after publication!"
```

[3] This facts reflects the simplified assumption that the impact factor of a scientist may not sink but only raise. More complicated assumptions are possible.

```
    next SetSubjectAttribute("ImpactFactor",IFrating, self)
  endseqblock
endseqblock
```

In case of a contribution to the screening data, a contribution attribute is set to *true* for the researcher in question:

```
  seqblock
    r2 <- CreateRandom
    if r2 > 0.9 then
      seq print "RESEARCHER " +Name(self)+ " Contributing
           data for screening center!"
   next SetSubjectAttribute("Contribution",true, self)
  endseqblock
```

The rest of the requester's program, i.e. the access request routines and the leave of a requester stay identical to the previous model.

Discussion: ABAC vs. TBAC?

As discussed in sections 2.2.3 and 2.2.4 both *attribute-based access control* and *trust-based access control* are suitable for dynamic coalition scenarios with high membership dynamics. However, it was argued that *TBAC* offers the more dynamic access control mechanism as, different from *ABAC*'s static attributes, it may deal with dynamic attribute changes over time, integrating some sense of experience into the trust evaluation. For example in the case study at hand it might be of interest to consider former interaction ratings in the access evaluation e.g. not forget very bad or very good ratings of the past. Whether this additional degree of dynamic access control is needed or not, depends on the actual use case and also on the design decision of the software designer.

In talks with medical personnel it became clear, that some kind of experience evaluation might be needed, allowing for a trust evaluation of a requester, not only through his static attributes, but also through integration of ratings of past interactions. Nevertheless, a final decision on how to eventually enforce the access control has not been made yet, and it was wished for a modeling of both approaches, in order to gain a better idea on how they function and compare their benefits.

In *ABAC* interaction ratings are to be stored as a single record, representing the last rating of an interaction with a requester, thereby "forgetting" all former interaction ratings. Another option would be to store each interaction rating as a single attribute. However, apart from the fact that this would possibly create a huge number of attributes, in *ABAC* there is no way to formulate access control rules over an average or some other kind of development curve of these ratings. Each entry would have to be treated one at a time.

With *TBAC* the "remembering" of former interaction records becomes simplified. Each rating could be stored in a set with all records carrying equal weight or different evolving weights over time, making recent entries "heavier" than older ones. When a requester wants to access data, an average over all ratings would be created according to the weights. How to create this average, would have to be defined in the trust evaluation operation, namely the *CalculateTrust()*-rule.

In the next section it will be shown how the research data exchange process may be modeled in a dynamic coalition model with *TBAC*, integrating some notion of experience through the usage of past interaction ratings. The calculation of an average interaction rating is kept as simple as possible, i.e. not implementing weights which change over time. Further refinement through adapting the *CalculateTrust()*-rule is left open for further refinement and implementation.

Enforcing Access Control through TBAC

In order to rate each interaction that took place, a function *Accessed()* has to count how many accesses took place, in order to let the *NHSC*-agent know how many accesses he has yet to rate[4]:

```
function Accessed : Agents -> SET
```

As explained in section 3.6.2 *CalculateTrust()*-rule has to be adapted for every use case at a time. In this case study the parameters that influence an agent's trust in another agent are: *interaction ratings* (an average over all recorded ratings will be calculated), an agent's *impact factor* and a *telephone rating* which is assigned only once when an agent has been contacted. Furthermore, the fact that an agent committed a *data contribution* to the screening data repository raises his trust to a high level immediately. An according *CalculateTrust()*-rule is shown below. *sum()* and *size()* are *CoreASM*-rules of the *Number*-background and of the *set*-background, returning the sum of a set of numbers or the number of elements in one set respectively. Since in this example all ratings are numbers between 0 and 5 the trust calculation adds up the three possible ratings, and then divides the resulting number through 15, resulting in a trust number between 0 and 5 respectively:

```
rule CalculateTrust(ag1, ag2) = {
  if DataContribution(ag2) then
    Trust(ag1,ag2):= 0.9
  else
  seqblock
    if not(InteractionRatings(ag1,ag2) = {}) then
      InteractionRatingAverage:=  sum(InteractionRatings(ag1,ag2)) /
                                size(InteractionRatings(ag1,ag2))
    else
      InteractionRatingAverage:= 0
      Trust(ag1,ag2):= (ImpactFactor(ag2) + InteractionRatingAverage
               + TelRating(ag1,ag2)) / 15
  endseqblock
}
```

The *NHSC* agent's program from 4.2.2 will then have to be adapted in the following way to integrate the *TBAC* mechanisms: After creation of the information, permissions have to be set up, connecting the access decision to a certain trust value which will be minimally needed:

```
rule NHSCProgram = {
  if AgentInfo(self) = {} then
  seqblock
    CreateInfo(self,"HighConfidentialData")
    CreateInfo(self,"MediumConfidentialData")
    CreateInfo(self,"LowConfidentialData")

    SetUpPermission(self,CallByName("HighConfidentialData"),read)
    SetTrustReq(self,1.0,
    PermissionID(CallByName("HighConfidentialData"),read))
```

[4]It was decided to record the amount of accesses that took place not in form of a *number* but in form of a *set*. This is due to the fact, that this function is foreseen to be updated in parallel update by various agents. Opposed to the *set*-background the *number*-background allows no parallel updates of the associated elements.

```
  SetUpPermission(self,CallByName("MediumConfidentialData"),read)
  SetTrustReq(self,0.7,
  PermissionID(CallByName("MediumConfidentialData"),read))

  SetUpPermission(self,CallByName("LowConfidentialData"),read)
  SetTrustReq(self,0.5,
  PermissionID(CallByName("LowConfidentialData"),read))
endseqblock
```

When an agent is called, the *NHSC* will assign an (in this example randomly created) rating number between 0 and 5 after which this trust parameter will be updated, which will finally result in an updated of the NHSC's trust in that agent:

```
choose a in CoalitionMembers(self)
with not  (Contacted(self,a)) do
seqblock
  print "NHSC: Calling RESEARCHER " +Name(a)+ "!"
  telrating <- CreateRandom
  telrating:= round(telrating*5)
  print "NHSC: Setting telephone rating to "+telrating+" after
       telephone call!"
  ChangeTrustParam(a,TelRating(self,a),telrating)
  Contacted(self,a):=true
endseqblock
```

The last lines of the new *NHSC*-program creates an *interaction rating* (again a randomly created number between 0 and 5) for former interactions for every agent and each interaction with the *NHSC*. This happens through adding the new *interaction rating* to the set of *interaction ratings* for this particular agent, eventually resulting in an update of the NHSC's trust in that agent. In the end the *Accessed()*-function counter will be reduced by one element, representing the conducted rating which has been crossed off the rating-to-do-list.

```
choose ag in CoalitionMembers(self)
with not(Accessed(ag)={}) do
  choose x in Accessed(ag) do
  seqblock
    irating <- CreateRandom
    irating := round(irating*5)
    print "NHSC: Adding interaction rating "+irating+" after
         interaction with requester "+Name(ag)
    ChangeTrustParam(ag,InteractionRatings(self,ag),
    InteractionRatings(self,ag) union {irating})
    remove x from Accessed(ag)
  endseqblock
```

The requester's programs has to be extended in so far as that every request results in the incrementing of the *Accesses()*-set. Thus, the following last line has to be added to the access request block of the requester's program:

```
choose action in "a","b","c" do
seqblock
  if action = "a" then
    seq print "RESEARCHER: " +Name(self)+ " Trying to access
```

```
          summary data!"
      next e <- RequestInfo(self, CallByName("NHSC"),
          CallByName("LowConfidentialData"), read)

    if action = "b" then
      seq print "RESEARCHER: " +Name(self)+ " Trying to access
          examination outcome!"
      next e <- RequestInfo(self, CallByName("NHSC"),
          CallByName("MediumConfidentialData"), read)

    if action = "c" then
      seq print "RESEARCHER: " +Name(self)+ " Trying to access
          Summary Data!"
      next e <- RequestInfo(self, CallByName("NHSC"),
          CallByName("HighConfidentialData"), read)

    if e=permit then add "x" to Accessed(self)
  endseqblock
```

Furthermore, publishing new scientific publications, as well as contributing data to the *NHSC* repository, will have to change the trust values in the obvious ways:

```
  seqblock
    r1 <- CreateRandom
    if r1 > 0.8 then
    seqblock
      print "RESEARCHER: " +Name(self)+ " Publishing scientific
          works!"
      IFrating <- CreateRandom
      IFrating:= round(IFrating*5)
      if IFrating > ImpactFactor(self) then
      seqblock
        print "RESEARCHER: " +Name(self9)+ " Setting impact factor
            to "+IFrating+" after publication!"
        ChangeTrustParam(self,ImpactFactor(self),IFrating)
      endseqblock
    endseqblock
  endseqblock

  seqblock
    r2 <- CreateRandom
    if r2 > 0.9 then
    seqblock
      print "RESEARCHER: " +Name(self)+ " Contributing data for
          screening center!"
      ChangeTrustParam(self,DataContribution(self),true)
   endseqblock
endseqblock
```

The rest of the requester's program, i.e. the leave of a requester stay identical to the basic dynamic coalition model. It is now possible to simulate the scenario with *attribute-based access control* as well as with *trust-based access control* as privacy-enforcing mechanisms. The next steps in this phase would consist of further refining the access control policies and the agent's programs.

4.3 Validation and Documentation Phase

In this section the validation and documentation phases for both case studies will be summarized.

4.3.1 Validation - Simulations and Interactive Feedback

As explained in section 3.7.2 a number of tools help with the validation of correctness requirements to the models. With these tools two things may be achieved: First, the output allows the modeler to validate the created program and helps him to spot unwanted or unexpected behavior. Second, the interaction with medical personnel which have little knowledge of software engineering is facilitated, allowing for a validation of the process outcome against the knowledge and requirements of actual domain experts.

Validation of the ASM Program through Simulation

The *CoreASM* environment allows the execution of *ASM* programs and through the input and output commands, such as *print*, even allows for interaction with the user. So in order to validate through simulation that the *ASM* program functions as expected a number of *print*-commands may create output that give information of the current state of the machine, resulting in an output protocol such as shown in figure 4.14 or in figure 4.16.

Through *print*-commands, at essential parts of the program the resulting output may give information about whether the program acted according to specification of not. For example, the *print*-command from the *ASM* model proposed in this case study:

```
print "ER: Monitoring patient!"
```

creates a corresponding output in the dialog window which indicates that the process has finished correctly. Thus, if this line does not occur and the program terminates before this lines is printed either expectedly through a shutdown command or unexpectedly through the occurrence of an error, the modeler will immediately know that his program does not work as expected. Through the clever use of these *print*-commands he may then debug the program until it lives up to his requirements.

Validation of the ASM Program with Medical Personnel

Through the simulation capabilities of the *CoreASM* environment and with help of the *State Exploration GUI* created in the course of this thesis (see section 3.7.2) medical personnel may be integrated into the validation step of the software engineering life cycle. The understanding of both the output protocol of an *ASM* program, as well as the visualization means of the *State Exploration GUI*, do not require any software engineering knowledge. Therefore, in the course of this case study both tools have been used to demonstrate the created models to medical personnel. In two meetings medical experts were asked for their input on the process, in order to validate that the model functions according to their expectations. They were also asked about how intuitive they found the graphical user interface as well as the output protocols. The experience of this interaction will be documented in the next section.

4.3.2 Documentation - Bridging Domains

Both models presented the case studies may be seen as an executable version of the *BPMN* of the stroke treatment process or the initial specification of a research data exchange platform for the *Newborn Hearing Screening Center* at *Charité Berlin* respectively. They therefore are another addition to the documentation of this process in a more executable and closer-to-implementation manner. The models at hand may be seen as a form of pseudo-code abstracting from implementation dependent aspects and focusing on the core of the algorithm or the process. Therefore, the model is not only easily understandable for software-engineers but with some introductory knowledge of the formalism it is also conceivable that domain experts from the medical field work with the model. Thus, the model may serve as a bridge between the two expert domains, trying to uniquely formalize the process in a common language that is abstract enough to be conceived by medical personnel but powerful enough to serve as an effective modeling tool for software modelers and developers.

4.4 Conclusions and Lessons Learned from the Case Studies

In this section the experience gained from modeling the real life stroke treatment scenario, as well as modeling the to-be-implemented research data exchange platform at the *Newborn Hearing Screening Center*, both located at at *Charité Berlin*, will be discussed. First, it will be reported, how much time it took to model the scenario with the framework proposed in this thesis.

4.4.1 Modeling Duration and Complexity

The process depicted in the *BPMN* (see section 4.1.1), although not exhaustively big, has some complex interactions of distributed agents and a number of critical conditions. Modeling these interactions in a basic dynamic coalition model and validating to some extent that the model works according to the *BPMN* definition took approximately half a work day, with the precondition that the modeler already had previous experience with the modeling practice in this framework.

Starting with this basic dynamic coalition model, it was only a matter of no more than an hour to identify the privacy-critical interactions and extend the model such as to gain access control mechanisms which constrain these interactions. In the course of this case study, first an *identity-based access control* mechanism was integrated and then after evaluating and validating the model with medical personnel it became evident that a role-based approach may be more appropriate, both for the access control as well as for the general interaction between agents. Therefore, another model was created as an extension of the basic dynamic coalition model in order to integrate *role-based access control* into the process. The creation and validation of this third model was not significantly longer than the second one.

Not counting the code of the framework models, the created *ASM* program codes were approximately 260, 280, 300 lines long with for the basic dynamic coalition model, the model with *IBAC* and the model with *RBAC* respectively. The slight raise of line number is explained through additional rule calls for creating access control policies, rules, roles, role assignments etc. Judging the relatively simple nature of the *BPMN* which forms the base of the *ASM* program, it can be said that the number of required lines seems relatively high. However, these numbers include all the *print*-commands which are not essential for the program as well as line breaks and blanks used for better visible comprehensibility of the program.

Without those elements, each of the models would be approximately 40 lines shorter. Furthermore, due to the simple nature of the *ASM* formalism, as well as the clear structure of the modeling framework, the complexity of the actual programming stays low.

The processes which would take place in the research data exchange platform which is planned for the *Newborn Hearing Screening Center* at *Charité Berlin* were less complicated to model. Modeling the main interactions in a basic dynamic coalition model and validating to some extent that the model works according to the needs of the *Screening Center* took no more than three hours, with the precondition that the modeler already had previous experience with the modeling practice in this framework.

Starting with this basic dynamic coalition model, it was only a matter of no more than an hour to identify the privacy-critical interactions and extend the model such as to gain access control mechanisms which constraint these interactions. In the course of this case study, both an *attribute-based access control* mechanism, as well as a *trust-based access control* approach, were tried out because the medical directors of the *Newborn Hearing Screening Center* have not yet made a final decision on how the access is to be controlled.

Not counting the code of the framework models, the created program codes were approximately 75, 105, 110 lines long for the basic dynamic coalition model, the model with *ABAC* and the model with *TBAC* respectively. The slight raise of line number is explained through additional rule calls for creating access control policies, attributes, permissions, trust values etc. However, these numbers include all the *print*-commands which are not essential for the program as well as line breaks and blanks used for better visible comprehensibility of the program. Without those elements, each of the models would be approximately 10 lines shorter. Furthermore, due to the simple nature of the *ASM* formalism, as well as the clear structure of the modeling framework, the complexity of the actual amount of programming efforts stays low.

4.4.2 Interaction with Medical Personnel

As mentioned in section 3.7.2 a graphical user interface has been created in the course of this thesis, which allows for the visual browsing of an *ASM* run after its execution. Together with the input and output capabilities of the *CoreASM* environment it allows for an easier accessibility to the *ASM* model making way for interdisciplinary interaction in the actual modeling process. In this spirit, medical domain experts have been integrated in the model validation process. In two meeting the output protocolls and the *State Exploration Visualization GUI* have been used to demonstrate the *ASM* program to the domain experts with the aim of receiving helpful input on the validity of the process as well as an opinion on what other features would be useful for modeling the coalitions at hand.

The experience of this case study shows that both tools help to keep the model as close to the actual domain specifications as possible. Both tools have been proven to be easily understandable by domain experts of the medical fields and returned provided valuable feedback from the latter: for example in the first meeting in the course of the first case study it became evident that a *role-based access control* is more suitable to the scenario at hand, leading to the creation of another model with *RBAC* as the access control mechanism.

One medical doctor assumed that, the proposed framework will be extremely useful for software projects in the medical field, since it allows for an integration of privacy aspects into the process modeling phase. According to him, other process modeling techniques like *BPMN* do not sufficiently respect these aspects.

4.4.3 Case Studies: Conclusion

The first case study of this thesis deals with the modeling of a real-life scenario at *Charité Berlin* and aims at formalizing the interaction of the various agents in this scenario from the view of a dynamic coalition with low to medium membership dynamics. For this purpose, first a basic dynamic coalition has been created which aims at implementing the *BPMN* model from section 4.1.1. In the next step, starting from the presumptions that the coalition membership is of low dynamics *identity-based access control* has been integrated. This model was then presented to medical personnel using output protocols and the *State Exploration Visualization GUI* (see section 3.7.2) in order to increase their understanding of the *ASM* program and gain more valuable insights on how to adapt the model. After the first simulations with the medical personnel it became clear that the dynamic coalition is more likely to be seen as a coalition with medium membership dynamics and therefore a model with *role-based access control*. After the creation of this third model in a second meeting it was again checked against medical specification and this time proved to be adequate and satisfying the domain experts requirements to a process model of this kind.

The second case study of this thesis specified a to-be-implemented research data exchange platform for the *Newborn Hearing Screening Center* at *Charité Berlin* and formalized the to-be-supported process from the view of a dynamic coalition with high membership dynamics. For this purpose, first a basic dynamic coalition has been created, which abstractly specifies the interaction which have to be supported. In the next steps *attribute-based access control*, as well as *trust-based access control*, have been integrated. Both resulting models have then been presented to the directors of the *Newborn Hearing Screening Center* at *Charité Berlin* who confirmed the correctness of the modeled process and agreed on the usefulness of the model for future implementations for modeling, validation and documentation purposes.

As a general conclusion it may be stated that the proposed framework is well suited to formally model, simulate and validate privacy-sensitive dynamic coalition processes in coalitions with low, medium and membership dynamics, supporting the software-engineering life cycle in its main phases: *requirement capture, detailed design, validation* and *documentation*. This statement is justified due to the fact, that both case studies constitute very different scenarios of dynamic coalitions with different membership dynamics. This fact suggests that the model may be applied in a wide field of dynamic coalition scenarios.

Thus, for the case studies at hand, the thesis of this work has been proven. Another experience gained from this case study is that the abstract nature of the used formalism, the input and output capabilities of *CoreASM*, as well as the *State Exploration GUI*, allowed for successful integration of medical personnel into the designing and validation process of the model, which made way for a more accurate depiction of the domain experts specification and satisfaction of their expectations.

Chapter 5

Conclusion and Outlook

In this chapter, the presented thesis will be summarized and concluded. Furthermore, an outlook over possible future research is presented.

5.1 Summary and Conclusion

The work at hand presents a formal modeling framework for privacy-aware dynamic coalitions. To this end, the *Abstract State Machine* formalism is equipped in order to define structure and operations of basic dynamic coalitions in general as well as privacy-aware dynamic coalitions in general. The latter are dynamic coalitions in which the information which is to be shared is regarded as privacy-sensitive.

Therefore, access control mechanisms which enforce privacy requirements are needed. Thus, common access control concepts like *identity-based access control* (*IBAC*), *role-based access control* (*RBAC*), *attribute-based access control* (*ABAC*) and *trust-based access control (TBAC)* mechanisms are formally integrated with the basic coalition model defining the "Dimension Access Control". Each of those mechanisms has a different application field each one defining coalition with different membership dynamics: *IBAC* is identified as being suitable for the privacy enforcement in coalitions with low membership dynamics. *RBAC* is regarded as being suitable for coalitions with medium membership dynamics and *ABAC* and *TBAC* are equipped in scenarios with high membership dynamics.

The thesis of this work states that the resulting modeling framework, consisting of a number of models according to the applied access control mechanism, supports the understanding and the development of software for these coalitions throughout the typical software engineering life cycle: *requirement capture*, *detailed design*, *validation* and *documentation*.

The correctness of this thesis as well as the framework itself is validated by means of two case studies taken from the medical field. The first case study constitutes a dynamic coalition process with low or medium membership dynamics according to the modeler's perspective. Therefore, three models have been created: one for the basic dynamic coalition and two for the definition of a privacy-sensitive dynamic coalition, one enforcing privacy through *IBAC* and the other through *RBAC* respectively. The second case study investigates a coalition with high membership dynamics and applies both *ABAC* and *TBAC* and discusses advantages and disadvantages of both approaches. The responsible personnel of both case studies have expressed their need for a dynamic approach to process modeling and software development in order to support their medical processes, which the work at hand aims at delivering.

Finally, this thesis ends with an outlook on possible further research, provides

an idea on how the framework may be used and extended in order to make way for *verification* of the formal models, as well as to improve the *interaction* with the user of the framework or to allow for using the framework for *model-driven development*.

The results of the application of the framework in the above mentioned case studies are promising: With help of this framework as well as a supporting *state visualization GUI* created during the course of this thesis, medical personnel were integrated effectively into the whole modeling and design process contributing with their domain expertise, throughout all four phases of the above mentioned life cycle.

> **Thus, the work at hand may be concluded with the statement, that it was shown how the thesis of this work holds for the presented case studies and how the framework presented here serves not only as a defining modeling concept for privacy-sensitive dynamic coalitions but also as a means to bridge the gap between software engineers and domain experts, such as the medical doctors in the presented case studies.**

5.2 Outlook and Further Research

In this section ideas for further research and possible application domains of the proposed framework of this thesis will be summarized. First, it will be discussed what steps would have to be taken to integrate some level of formal verification into the framework. Secondly, it will be shown how the framework, as well as its supporting applications, could be extended in order to increase its usability and usefulness.

5.2.1 Validation and Verification Approaches

In this thesis it was shown how the proposed framework in combination with *CoreASM* may function as a simulating environment for privacy-sensitive dynamic coalitions. A definitive next step in improving these simulation-based validation capabilities is the integration of sophisticated and established testing methods into the validation process: As the framework operates in a formal manner and allows for simulation as well as the extension through means of automated testing, the ground work for integrating professional testing processes into the framework is already given. However, in large scenarios a testing or simulation of every possible situation may be impractical or even impossible. In order to achieve a more formal approach to a verification of our model there may be different verification approaches, which may be studied and utilized in further researches in order to increase the applicability and usability of the framework.

The general concept of *ASMs* explicitly provides for a logic based structural analysis of the characteristics and properties of a given abstract state machine (compare with [BS03, GKOT00]). Such a logical consideration is accomplished by defining a certain logical formalism whose formulas are adequate for a description and comprising of the states and conditions of the *ASM*. In the normal case examples of suitable logics in which such *ASMs* can be comprised are first-order predicate logics or (with a limited usability) classical propositional logics. Furthermore certain extensions of such logics by temporal considerations appear to be most adequate to cover the dynamics which are apparent in the processes modeled in abstract state machines. A logic based verification models created with the framework of this thesis needs for two requirements: the first is that the model must be given in a certain *ASM* state-transition diagram; the second is the need for an adequate logical formalism whose formulas are expressive enough to state aspired privacy and

access control requirements. Here especially the work of Glausch and Reisig on a state-transition graph visualization of distributed *ASMs* (see [GR07]) seems to be an adequate starting point for the development of such a logical based verification approach.

Another possibility for a verification of the framework is given by the general ideas on model checking of *ASMs*. In the general case model checking refers to the verification of a system (which is given as a corresponding model of a certain logic) against a specified property of the system (which is given as a certain formula of the logic) (see for example [CGP00]). As a result for a given model M and a given formula φ a model checking algorithm is considered to calculate if φ is satisfied in M or not. There exists an application which allows for the testing of certain concrete requirements (given as statements of the *CoreASM*-language) in any given implementation in the *CoreASM*-language: *CoreASM2Promela* translates *CoreASM*-programs into the *Promela* language which in turn can be model checked by the Spin model checker (see [FGM, Ma07]). In this understanding the application cannot be seen as a *CoreASM*-model-checker, but as an external tool which transfers the model checking task of *ASM* to existing solutions. Although the application is supposed to allow for the consideration of nearly any expression stateable in the *CoreASM* language, the use of the `extend` command in our model cannot be translated into the *Promela* language by the application. This is due to the fact that the `extend` command allows for the extension of our workspace by arbitrary new variables. An uncontrolled or not limited introduction of new variables of this kind may lead to an uncontrolled expansion of states to be considered by a model checker, which in turn may finally lead to an abortion of the algorithm. There are two feasible solutions to this problem: the first is to extend the model checker application to allow for the resolving of the `extend` command; the second is to change our model in a way which avoids the usage of the `extend` command. As in the normal case in concrete *ASMs* there is no unlimited usage of `extend`, it appears to be a possible task to extend the translation of *CoreASM* code into *Promela*-code by a certain (maybe constricted) use of `extend`.[1] Such an extension could be subject for future works following the outlined ideas for such an extension as given in [Ma07].

As mentioned in the introduction of this thesis, "verification" was not considered in the work at hand, as the problems in realizing verification are not regarded to be in the scope of this thesis' motivation. Furthermore, its realization may be a complex task which in the view of the author of this thesis is worthy for at least one dissertation on its own.

5.2.2 Improving Interaction of the User with the Framework

A critical aspect of the applicability of the proposed modeling framework, lies in the possibilities of interaction with the user, which could be the modeler or a domain expert who is suppose to take part in tests and simulations. A number of ideas came up during the course of this thesis, each having as a goal the improvement of the interactive capabilities of the framework:

Utilizing Input-Parameters

CoreASM allows not only for output-messages through the *print*-command, but is also equipped with an *input*-rule, allowing for direct interaction with a user during runtime. This feature might be utilized for interactive simulations, e.g. allowing for the creation of agents, dynamic coalitions, access control policies and even whole

[1] Although not further mentioned there is another *CoreASM* rule which cannot be translated into *Promela* up to now. This rule is the `import` rule.

processes during *ASM* runtime. The creation of such an *ASM* program would have to have a big number of conditions based on the actual given input and would therefore result in a much bigger complexity. However, the benefits of integrating domain experts not only as observers but as actors into the simulation process may be worth the effort.

GUI for Creating Dynamic Coalition Programs

Another idea envisions a creation of *ASM* programs not through actual coding but through a graphical user interface which allows to use model functions in a point and click manner. For example, the programs for each agents could be created by dragging and dropping the model rules into the execution program of an agent, instantiating them with existing elements. This approach would reduce the necessary period of vocational adjustment and theoretically even allow non-software-engineers to create, test and simulate dynamic coalition models.

File-Input for Importing Scenario Data

In order to improve statistical testing, or the simulation of information exchange with actual, real-life data the *CoreASM-input*-command could be extended so that it allows for the integration of external data files. While the actual usage and integration of such data into a dynamic coalition model is scenario dependent and up to the modelers design-decisions, extending the *input*-rule to allow for certain text-files to be imported is a software-technical issue, which should be straightforward to realize and could be seen as a contribution to the *CoreASM* tool itself.

Integration of Environment Behavior

In order to improve simulation capabilities Altenhofen and Farahbod [AF10] have proposed a plugin which offers a scripting language which is able to define so called *scenarios*, meaning scripts for the behavior of an external environment with which the *ASM* specification may interact. With help of scenario-specification, they envision the testing of *ASM*-specifications as black-boxes or grey-boxes, in order to check for their fitness for various environments. Integrating this approach with the framework presented in this thesis could be subject to future research, as well as the application of this concept in dynamic coalition case studies.

5.2.3 Using the Framework for Model-Driven Development

The proposed model, no matter how basic or abstract, is eventually supposed to be utilized in real-life software-engineering projects. This section will show how with further research contributions its usefulness for actual dynamic coalition implementations may be improved.

Executable Code for DCs in ASM

The models created with the framework at hand, constitute distributed agent scenarios in which the behavior of each acting agent is specified in terms of *ASM*. In [BS03] it was shown, how through further refinement *ASM* specifications may be detailed to such an extend, that they may actually serve as initial input for a program skeleton in an actual programming language. For this purpose a translating tool would have to be created which gets an dynamic coalition *ASM* specification as input and translates it to a programming language of choice, resulting in an executable software implementation based on the *ASM* specification. Through this tool

the benefits of model-driven development may be won, for example making sure, that the software implementation corresponds to the validated *ASM* specification.

Generating Access Control Policies

As mentioned above, this framework's model components for *identity-based access control* (*IBAC*), *role-based access control* (*RBAC*) and *attribute-based access control* (*ABAC*) are closely connected to the XACML-standard for *IBAC* and *ABAC* and the *NIST*-standard for *RBAC* respectively. Therefore it is conceivable, to take the access control policy definitions of a dynamic coalition *ASM* specification as an input and similarly to the model-driven development approach translate them into actual policy skeletons of the according standard. For *trust-based access control* (*TBAC*) no such standard exists yet. creating such a standard would exceed the capacities of any single researcher but could very well be subject for a future research project on a bigger scale.

List of Figures

Bibliography

[AD05] W.J. Adams and IV Davis, N.J. Toward a decentralized trust-based access control system for dynamic collaboration. In *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, pages 317–324, 2005.

[AF10] Michael Altenhofen and Roozbeh Farahbod. Bârun: A scripting language for coreasm. In Marc Frappier, Uwe Glässer, Uwesser, Sarfraz Khurshid, RÃ©gine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, volume 5977 of *Lecture Notes in Computer Science*, pages 47–60. Springer Berlin Heidelberg, 2010.

[AK06] Ali E. Abdallah and Etienne J. Khayat. Formal Z Specifications of Several Flat Role-Based Access Control Models. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, SEW '06, pages 282–292, Washington, DC, USA, 2006. IEEE Computer Society.

[AMCGR05] Florina Almenárez, Andrés Marín, Celeste Campo, and Carlos García R. Trustac: trust-based access control for pervasive devices. In *Proceedings of the Second international conference on Security in Pervasive Computing*, SPC'05, pages 225–238, Berlin, Heidelberg, 2005. Springer-Verlag.

[AMCR04] Florina Almenárez, Andrés Marín, Celeste Campo, and Carlos García R. PTM: A Pervasive Trust Management Model for Dynamic Open Environments. In *In: First workshop on pervasive security, privacy and trust PSPT'04 in conjunction with MOBIQUITOUS*, 2004.

[BF07] Jeremy Bryans and John Fitzgerald. Formal Engineering of XACML Access Control Policies in VDM++. In Michael Butler, Michael Hinchey, and María Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 37–56. Springer Berlin / Heidelberg, 2007.

[BF08] J. W. Bryans and J. S Fitzgerald. The Verifiable Virtual Organisation: A Position Paper. In *Proceedings of Formal Aspects of Virtual Organisations 2008*, pages 6–15, 2008.

[BFG+08] J. W. Bryans, J. S. Fitzgerald, D. Greathead, C. B. Jones, and R. J. Payne. A Dynamic Coalitions Workbench: Final Report. Technical report, Newcastle University, 2008.

[BFJ+06] J. W. Bryans, J. S. Fitzgerald, C. B. Jones, I. Mozolevsky, Jeremy W. Bryans, John S. Fitzgerald, Cliff B. Jones, and Igor Mozolevsky. Dimensions of dynamic coalitions. Technical report, 2006.

[BFJM06] Jeremy Bryans, John S. Fitzgerald, Cliff B. Jones, and Igor Mozolevsky. Formal Modelling of Dynamic Coalitions, with an Application in Chemical Engineering. Technical report, 2006.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 164–173. IEEE, 1996.

[BFP06] J. W. Bryans, J. S. Fitzgerald, and P. Periorellis. Model Based Analysis and Validation of Access Control Polcies. Technical report, Newcastle University, School of Computing Science, 2006.

[BFRRM09] Laura Bocchi, José Luiz Fiadeiro, Noor Rajper, and Stephan Reiff-Marganiec. Structure and Behaviour of Virtual Organisation Breeding Environments. In *FAVO*, pages 26–40, 2009.

[BG94] Andreas Blass and Yuri Gurevich. Evolving Algebras and Linear Time Hierarchy, 1994.

[BG03] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic (TOCL)*, 4(4):578–651, 2003.

[BGK+05] P. L. Brantingham, U. Glässer, B. Kinney, K. Singh, and M. Vajihollahi. Modeling Urban Crime Patterns: Viewing Multi-Agent Systems as Abstract State Machines. In *PROC.ASM05 Université de Paris 12*, 2005.

[BGRR07] Andreas Blass, Yuri Gurevich, Dean Rosenzweig, and Benjamin Rossman. Interactive Small-Step Algorithms I: Axiomatization. *CoRR*, abs/0707.3782, 2007.

[Bry05] Jery Bryans. Reasoning about XACML policies using CSP. In *Proceedings of the 2005 workshop on Secure web services*, SWS '05, pages 28–35, New York, NY, USA, 2005. ACM.

[BS03] Egon Börger and Robert Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer Verlag, 2003.

[BS11a] Sebastian Bab and Nadim Sarrouh. Formale Modellierung von Access-Control-Policies in Dynamischen Koalitionen. In *GI Proceedings, Informatik 2011 - Informatik schafft Communities*, page 402, 2011.

[BS11b] Sebastian Bab and Nadim Sarrouh. Towards a formal model of privacy-sensitive dynamic coalitions. In Jeremy Bryans and John S. Fitzgerald, editors, *FAVO*, volume 83 of *EPTCS*, pages 10–21, 2011.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[CGvH+12] Davide Ceolin, Paul T. Groth, Willem Robert van Hage, Archana Nottamkandath, and Wan Fokkink. Trust Evaluation through User Reputation and Provenance Analysis. In Fernando Bobillo, Rommel N. Carvalho, Paulo Cesar G. da Costa, Claudia d'Amato, Nicola Fanizzi, Kathryn B. Laskey, Kenneth J. Laskey, Thomas Lukasiewicz, Trevor Martin, Matthias Nickles, and Michael Pool, editors, *URSW*, volume 900 of *CEUR Workshop Proceedings*, pages 15–26. CEUR-WS.org, 2012.

[CR06] Sudip Chakraborty and Indrajit Ray. TrustBAC: integrating trust relationships into the RBAC model for access control in open systems. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, SACMAT '06, pages 49–58, New York, NY, USA, 2006. ACM.

[CW13] Ed Coyne and Timothy R. Weil. ABAC and RBAC: Scalable, Flexible, and Auditable Access Management. *IT Professional*, 15(3):14–16, 2013.

[EA13] Sergio Esparcia and Estefanía Argente. Defining Virtual Organizations Following a Formal Approach. In Joaquim Filipe and Ana Fred, editors, *Agents and Artificial Intelligence*, volume 271 of *Communications in Computer and Information Science*, pages 365–381. Springer Berlin Heidelberg, 2013.

[EN10] Florian Eilers and Uwe Nestmann. Deriving trust from experience. In Pierpaolo Degano and JoshuaD. Guttman, editors, *Formal Aspects in Security and Trust*, volume 5983 of *Lecture Notes in Computer Science*, pages 36–50. Springer Berlin Heidelberg, 2010.

[Far09] Roozbeh Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, Simon Fraser University, 2009.

[FGG07] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1):71–103, 2007.

[FGM] Roozbeh Farahbod, Uwe Glässer, and George Ma. Model Checking CoreASM Specifications.

[FK92] David Ferraiolo and Richard Kuhn. Role-Based Access Control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[FL09] John S. Fitzgerald and Peter Gorm Larsen. *Modelling Systems - Practical Tools and Techniques in Software Development (2. ed.)*. Cambridge University Press, 2009.

[FLM+05] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs For Object-oriented Systems*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2005.

[Gam88] Diego Gambetta, editor. *Trust: making and breaking cooperative relations*. Basil Blackwell, New York, NY [u.a.], 1988.

[GH93] Yuri Gurevich and James K. Huggins. The Semantics of the C Programming Language. In *Computer Science Logic, volume 702 of LNCS*, pages 274–308. Springer, 1993.

[GKOT00] Yuri Gurevich, Philipp W. Kutter, Martin Odersky, and Lothar Thiele, editors. *Abstract State Machines, Theory and Applications, International Workshop, ASM 2000, Monte Verità, Switzerland, March 19-24, 2000, Proceedings*, volume 1912 of *Lecture Notes in Computer Science*. Springer, 2000.

[GR07] Andreas Glausch and Wolfgang Reisig. An ASM-Characterization of a Class of Distributed Algorithms. In *Proceedings of the Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis*, Festschrift volume of Lecture Notes in Computer Science. Springer, 2007. to appear.

[Gur93] Yuri Gurevich. Evolving algebras: An attempt to discover semantics, 1993.

[Gur00] Yuri Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Trans. Comput. Logic*, 1(1):77–111, July 2000.

[HCA+09] Ali Nasrat Haidar, P. V. Coveney, Ali E. Abdallah, Peter Y. A. Ryan, B. Beckles, J. M. Brooke, and M. A. S. Jones. Formal Modelling of a Usable Identity Management Solution for Virtual Organisations. In *FAVO*, pages 41–50, 2009.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[HW02] James K. Huggins and Charles Wallace. An Abstract State Machine Primer. Technical report, 2002.

[JT52] Bjarni Jónnson and Alfred Tarski. Boolean algebras with operators. *American Journal of Mathematics*, 74(1):127–162, 1952.

[Kat98] B.R. Katzy. Design and implementation of virtual organizations. In *System Sciences, 1998., Proceedings of the Thirty-First Hawaii International Conference on*, volume 4, pages 142–151 vol.4, 1998.

[KHP07] Vladimir Kolovski, James Hendler, and Bijan Parsia. Analyzing web access control policies. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 677–686, New York, NY, USA, 2007. ACM.

[Klu98] R. Klueber. A framework for virtual organizing. In *VoNet Workshop*, 1998.

[KM10] Hristo Koshutanski and Antonio Maña. Interoperable semantic access control for highly dynamic coalitions. *Security and Communication Networks*, 3(6):565–594, 2010.

[KMPP] Manuel Koch, Luigi V. Mancini, and Francesco Parisi-Presicce. A Formal Model for Role-Based Access Control using Graph Transformation. In *In Proc. of 5th ESORICS, volume 1895 of LNCS*, pages 122–139. Springer.

[Lee11] Adam J Lee. Credential-Based Access Control. In *Encyclopedia of Cryptography and Security*, pages 271–272. Springer, 2011.

[Let01] Nick Lethbridge. An I-Based Taxonomy of Virtual Organisations and the Implications for Effective Management. *Informing Science*, 4 No 1, 2001.

[Lin06] Hakan Lindqvist. Mandatory access control. *Master's Thesis in Computing Science, Umea University, Department of Computing Science, SE-901*, 87, 2006.

[LM03] Ninghui Li and John C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages*, January 2003. To appear.

[Luh00] Niklas Luhmann. Familiarity, confidence, trust: Problems and alternatives. *Trust: Making and breaking cooperative relations*, 6:94–107, 2000.

[LWR09] Min Li, Hua Wang, and David Ross. Trust-Based Access Control for Privacy Protection in Collaborative Environment. In *Proceedings of the 2009 IEEE International Conference on e-Business Engineering*, ICEBE '09, pages 425–430, Washington, DC, USA, 2009. IEEE Computer Society.

[LZK05] Hermann Löh, Chunyan Zhang, and Bernhard Katzy. Modeling for Virtual Organizations. In LuisM. Camarinha-Matos, Hamideh Afsarmanesh, and Martin Ollus, editors, *Virtual Organizations*, pages 29–43. Springer US, 2005.

[Ma07] George Ma. Model checking support for CoreASM: Model checking distributed abstract state machines using spin. Technical report, 2007.

[MPT12] Massimiliano Masi, Rosario Pugliese, and Francesco Tiezzi. Formalisation and implementation of the XACML access control mechanism. In *Proceedings of the 4th international conference on Engineering Secure Software and Systems*, ESSoS'12, pages 60–74, Berlin, Heidelberg, 2012. Springer-Verlag.

[MST09] Jarred McGinnis, Kostas Stathis, and Francesca Toni. A Formal Framework of Virtual Organisations as Agent Societies. In *FAVO*, pages 1–14, 2009.

[NSM07] M.R. Nami, M. Sharifi, and A. Malekpour. A Preliminary Formal Specification of Virtual Organization Creation with RAISE Specification Language. In *Software Engineering Research, Management Applications, 2007. SERA 2007. 5th ACIS International Conference on*, pages 227–232, 2007.

[oST09] National Instiute of Standards and Technology. A Survey of Access Control Models, 2009.

[rcsff01] robert c. solomon and fernando flores. *building trust in business, politics, relationships, and life*. Oxford University Press, 2001.

[RMR11] Stephan Reiff-Marganiec and NoorJ. Rajper. Modelling Virtual Organisations: Structure and Reconfigurations. In LuisM. Camarinha-Matos, Alexandra Pereira-Klen, and Hamideh Afsarmanesh, editors, *Adaptation and Value Creating Collaborative Networks*, volume 362 of *IFIP Advances in Information and Communication Technology*, pages 297–305. Springer Berlin Heidelberg, 2011.

[SA13] M. Shadi and H. Afsarmanesh. Behavior modeling in virtual organizations. In *Advanced Information Networking and Applications Workshops (WAINA), 2013 27th International Conference on*, pages 50–55, 2013.

[Sar13] Nadim Sarrouh. Formal modeling of trust-based access control in dynamic coalitions. In *COMPSAC Workshops*, pages 224–229. IEEE, 2013.

[SBC+97] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles Youman. The ARBAC97 Model for Role-Based Administration of Roles: Preliminary Description and Outline, 1997.

[SFK00] Ravi Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role-based access control: towards a unified standard. In *Proceedings of the fifth ACM workshop on Role-based access control*, RBAC '00, pages 47–63, New York, NY, USA, 2000. ACM.

[SHYL06] Y.L. Sun, Zhu Han, Wei Yu, and K.J.R. Liu. A trust evaluation framework in distributed networks: Vulnerability analysis and defense against attacks. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–13, 2006.

[SL05] William Song and Xiaoming Li. A Conceptual Modeling Approach to Virtual Organizations in the Grid. In Hai Zhuge and GeoffreyC. Fox, editors, *Grid and Cooperative Computing - GCC 2005*, volume 3795 of *Lecture Notes in Computer Science*, pages 382–393. Springer Berlin Heidelberg, 2005.

[SLW98] Y.P. Shao, S.Y. Liao, and H.Q. Wang. A model of virtual organisations. *Journal of Information Science*, 24:305Ñ–312, June 1998.

[TB06] G. Theodorakopoulos and J.S. Baras. On trust models and trust evaluation metrics for ad hoc networks. *Selected Areas in Communications, IEEE Journal on*, 24(2):318–328, 2006.

[Tur36] Alan M Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 42(2):230–265, 1936.

[VVE10] Toby Velte, Anthony Velte, and Robert Elsenpeter. *Cloud Computing, A Practical Approach*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010.

[Win05] Lyle J. Winton. A simple virtual organisation model and practical implementation. In *Proceedings of the 2005 Australasian workshop on Grid computing and e-research - Volume 44*, ACSW Frontiers '05, pages 57–65, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[ZCW+07] Saad Zafar, Robert Colvin, Kirsten Winter, Nisansala Yatapanage, and R. G. Dromey. Early Validation and Verification of a Distributed Role-Based Access Control Model. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference*, APSEC '07, pages 430–437, Washington, DC, USA, 2007. IEEE Computer Society.

[ZQG04] Yuqing Zhai, Yuzhong Qu, and Zhiqiang Gao. Agent-Based Modeling for Virtual Organizations in Grid. In Hai Jin, Yi Pan, Nong Xiao, and Jianhua Sun, editors, *Grid and Cooperative Computing - GCC 2004 Workshops*, volume 3252 of *Lecture Notes in Computer Science*, pages 83–89. Springer Berlin Heidelberg, 2004.

[ZTS+08] Mikolaj Zuzek, Marek Talik, Tomasz Swierczynski, Cezary Wisniewski, Bartosz Kryza, Lukasz Dutka, and Jacek Kitowski. Formal Model for Contract Negotiation in Knowledge-Based Virtual Organizations. In Marian Bubak, G. Dick van Albada, Jack Dongarra, and Peter M. A. Sloot, editors, *ICCS (3)*, volume 5103 of *Lecture Notes in Computer Science*, pages 409–418. Springer, 2008.