UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Software Engineering

Olgun Cakabey

Role-Based Access Control Using Knowledge Acquisition in Automated Specification

Master Thesis (30 ECTS)

Supervisor: Dr. Raimundas Matulevičius

Author:	Olgun Cakabey	"……. "	Nov 2012
Supervisor:	Raimundas Matulevičius	"……..."	Nov 2012
Approved for defence			
Professor:	Marlon Dumas	"……"	Nov 2012

TARTU 2012

ABSTRACT

Security is considered to be an aspect of information systems. Role-based access control (RBAC) is an approach to restricting system access to authorized users in information systems. Existing security modeling languages and/or approaches address the security of the IS, however existing languages or approaches do not necessarily conforms to the needs of RBAC. There are several modeling languages (e.g. SecureUML, UMLSec, etc.) to represent RBAC but they are not interoperable and it is not easy to compare one with another. Each modeling language represents different perspectives on information systems. Besides, there is a need to merge design and requirement stages in order to discover system security concerns and analyze related security trade-offs at the earlier stages. Knowledge acquisition in automated specification (KAOS) is a goal oriented requirement engineering approach to elicit software requirements. In this point, KAOS will be a key solution in order to combine requirements with design principles.

In this thesis, we will analyze KAOS to apply RBAC. More specifically, we will apply a systematic approach to understand how KAOS can be used to apply RBAC. Our research work will be based on the transformation rules between KAOS-SecureUML and KAOS-UMLSec, and vice versa. Moreover, through these transformations we will show how we aligned KAOS to RBAC.

The contribution of this research has several benefits. Firstly, it will potentially help to understand how KAOS could deal with RBAC. Secondly it will define the approach to elicit security requirements for RBAC at early stages of the IS development. This will apply our results in a case study to measure the correctness of the defined approach. Thirdly, the transformations from/to the KAOS would help IS developers and the other system stakeholders (e.g. system analysts, system administrators, etc.) to understand how important these security approaches (KAOS, SecureUML and UMLSec) are and which one has more advantages/disadvantages. We plan to validate our results for transformation rules and the models regarding their correctness that will be measured. Last but not least, we will be able to justify the design stage with requirement stage.

ACKNOWLEDGMENTS

I am heartily thankful to my supervisor, Raimundas Matulevičius, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

Lastly, I offer my regards and blessings to my family, Latif Cakabey, Gülben Cakabey and Basak Cakabey and my colleagues who supported me in any respect during the completion of the project.

Olgun Cakabey

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATION

ABAC : Attribute-Based Access Control
AC : Authorization Constraints
AT : Associated Tags
DAC : Discretionary Access Control
IS : Information System
KAOS : Knowledge Acquisition in Automated Specification
KS : KAOS to SecureUML
KU : KAOS to UMLSec
MAC : Mandatory Access Control
RBAC : Role-Based Access Control
SK : SecureUML to KAOS
UK : UMLSec to KAOS
UML : Unified Modeling Language

Chapter 1. Introduction

Nowadays information systems are everywhere in our lives, such as banking, education, health, and legacy, etc., therefore information system security plays an important role in information systems, which in many cases, information is confidential and should not be accessible to everyone.

In computer systems security, role-based access control (RBAC) [11] is an approach to restricting system access to authorized users. RBAC is a very popular security pattern in information systems. It is basically used for ensuring confidentiality. Knowledge acquisition in automated specification (KAOS) [13] is a goal-oriented software requirements capturing approach in requirements engineering.

Even though security is an important aspect in information systems, security issues and concerns are raised only when the system is already in use, or is about to start running, or luckily in the best case, security is just considered during the late system development phases for instance implementation phase. This is an obstacle to secure the system development. System security concerns should be discovered and related security trade-offs should be analyzed at the earlier stages such as requirements or design stages. There is possible way to guide such an analysis is suggested by the model-driven approaches. For example, "SecureUML [7] and UMLSec [7] which are both originated from UML and also deal with security modeling, these modeling approaches could be applied to model RBAC in a system [9], they are rather specific than general. They actually both contain targeted concepts for RBAC" but they fail to satisfy our needs about the security analysis in the earlier stages. Our motivation to look at KAOS was also strengthened by the fact that it is not used to analyze the access control before. However, it contains basic RBAC concepts. In addition to this, we can justify the design with the requirements; this means we can secure the system development at early stages. In order to continue our research, we formulate the following research question:

Can KAOS be aligned to model RBAC or not? If yes, how?

In order to answer this research question we have analyzed the KAOS literature [12] and tested it on Meeting Scheduler Example [16] and Food Delivery Example. At the same time, we also benefit from the SecureUML and UMLSec literature work [7] and application of these approaches on Meeting Scheduler Example. We had chance to compare each modeling approaches. Our observations are that KAOS can be applied to model RBAC. According to the common concepts and the similarities between KAOS and RBAC, we can use KAOS to define RBAC.

The structure of the thesis is as follows: after giving brief introduction in Chapter 1. As Background part, in Chapter 2 we introduce the general RBAC model. In Chapter 3 we introduce one modeling approach - KAOS model. Later on, in Chapter 4 we present related work that has been done with RBAC on two modeling approaches – SecureUML and UMLSec. As Contribution part, in Chapter 5 we understand the KAOS language with its similarities with RBAC. Then, in Chapter 6 and Chapter 7 we define transformation rules between KAOS-SecureUML and KAOS-UMLSec and vice versa. As Validation part, in Chapter 8 we select one of the validation options and design our test for validity. In Chapter 9 we situate our contribution in a case study and discuss our results. Finally as Conclusion part, in Chapter 10 we finalize our work and present some future work.

PART I

BACKGROUND

In Background Part, we are going to present literature work and reviews of RBAC, KAOS, and related works.

Mainly on RBAC side, we focus on RBAC reference models, and other access control models. On KAOS side, we discuss about abstract syntax, concrete syntax and semantics of KAOS, KAOS models and relationship between security and KAOS. After that, we finalize with related works that has been done with RBAC on different security modeling languages.

Chapter 2. Role-based Access Control

"In computer systems security, role-based access control (RBAC) is an approach to restricting system access to authorized users. Within an organization, roles are created for various job functions. The permissions to perform certain operations are assigned to specific roles. Members of staff (or other system users) are assigned particular roles, and through those role assignments acquire the computer permissions to perform particular computer-system functions. Since users are not assigned permissions directly, but only acquire them through their role(s), management of individual user rights becomes a matter of simply assigning appropriate roles to the user's account; this simplifies common operations, such as adding a user, or changing a user's department." [1] and [17]. In Fig. 1 we see the basic elements of RBAC model.

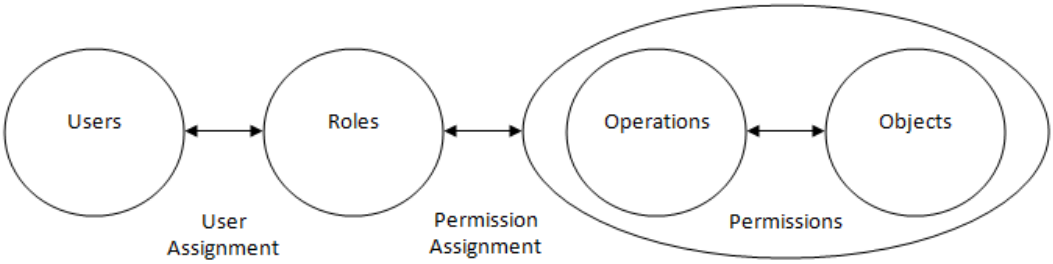

Figure 1 RBAC

(adapted from [1])

2.1. Terms and Concepts

The Table 1 shows the terms and concepts of RBAC [11]. This table covers all the terms and concepts of RBAC but actually we are not going to use and benefit from all of them. Users, Roles, Operations, Objects and Permissions are the main elements of RBAC.

2.2. RBAC Reference Models

In this section, we are going to present RBAC reference models. The figure 2 shows the family of RBAC models.

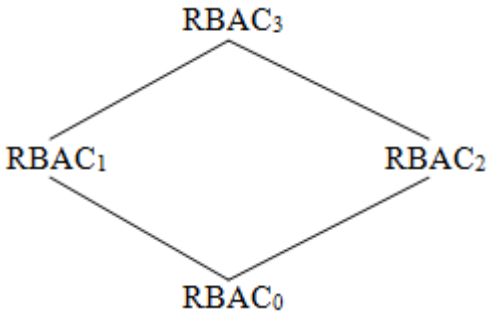

Figure 2 A Family of RBAC Models

(adapted from [11])

Table 1 RBAC Terms and Concepts

(adapted from [11])

Access	A specific type of interaction between a subject and an object that results in the flow of information from one to the other.
Access control	The process of limiting access to the resources of a system only to authorized programs, processes, or other systems.
Administrative role	A role that includes permission to modify the set of users, roles, or permissions, or to modify the user assignment or permission assignment relations.
Constraint	A relationship between or among roles.
Group	A set of users.
Object	A passive entity that contains or receives information.
Permissions	A description of the type of authorized interactions a subject can have with an object.
Resource	Anything used or consumed while performing a function. The categories of resources are time, information, objects, or processors.
Role	A job function within the organization that describes the authority and responsibility conferred on a user assigned to the role.
Role hierarchy	A partial order relationship established among roles.
Session	A mapping between a user and an activated subset of the set of roles the user is assigned to.
Subject	An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state.
System administrator	The individual who establishes the system security policies, performs the administrative roles, and reviews the system audit trail.
User	Any person who interacts directly with a computer system.

2.2.1. $RBAC_0$

$RBAC_0$ is also called base model or core model. It has four entities: users, roles, permissions and sessions.

Users and roles: User is a human and role is a job function within the organization which describes the authority and responsibility of a member.
Permissions: Permission is an ability to access to one or multiple objects in the system. As a term, authorization, access right or privilege are also used instead of permission in the literature.
Sessions: Session is a mapping between the user and a subset of the roles belong to that user.

$RBAC_0$ model has the following components:

- "*U*, *R*, *P*, and *S* respectively represent users, roles, permissions, and sessions;
- $PA \subseteq P \times R$, a many-to-many permission-to-role assignment relation;
- $UA \subseteq U \times R$, a many-to-many user-to-role assignment relation;
- user: $S \rightarrow U$, a function mapping each session s_i to the single user user(s_i) (constant for the session's lifetime); and

- roles: $S \rightarrow 2^R$, a function mapping each session s_i to a set of roles roles(s_i) $\subseteq$ {r | (user(s_i), r) $\in$ *UA*} (which can change with time) and session s_i has the permissions $\cup_r \in_{roles(si)}$ {*p* | (*p, r*) $\in$ *PA*}" [11].

2.2.2. RBAC$_1$

$RBAC_1$ is also called hierarchical RBAC. It introduces role hierarchies.

$RBAC_1$ model has the following components:

- "*U*, *R*, *P*, *S*, *PA*, *UA*, and *user* are unchanged from $RBAC_0$;
- *RH* $\subseteq$ *R* X *R* is a partial order on *R* called the role hierarchy or role dominance relation, also written as $\geq$; and
- *roles*: $S \rightarrow 2^R$ is modified from $RBAC_0$ to require
  roles(s_i) $\subseteq$ {*r* | ($\exists r' \geq r$) [(*users*(s_i), r') $\in$ *UA*]} (which can change with time] and session s_i has the permissions
  $\cup_r \in_{roles(si)}$ {*p* | $\exists r'' \leq r$) [(*p*, *r*") $\in$ *PA*]}" [11].

2.2.3. RBAC$_2$

$RBAC_2$ is also called constrained RBAC. It introduces constraints.

"$RBAC_2$ is unchanged from $RBAC_0$ except for requiring that there be constraints to determine the acceptability of various components of $RBAC_0$. Only acceptable values will be permitted" [11].

2.2.4. RBAC$_3$

$RBAC_3$ is also called consolidated model. It provides both role hierarchies and constraints, as it combines $RBAC_1$ and $RBAC_2$.

2.3. Other Access Control Models

There are four most widely recognized models. First one is RBAC which we already gave its description and characteristics. The others are Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Attribute-Based Access Control (ABAC).
Access control models are sometimes categorized as either discretionary or non-discretionary. For example, MAC and RBAC are non-discretionary. "Role-based access control (RBAC) is an access policy determined by the system, not the owner. RBAC is used in commercial applications and also in military systems, where multi-level security requirements may also exist. RBAC differs from DAC in that DAC allows users to control access to their resources, while in RBAC, access is controlled at the system level, outside of the user's control. Although RBAC is non-discretionary, it can be distinguished from MAC primarily in the way permissions are handled. MAC controls read and write permissions based on a user's clearance level and additional labels. RBAC controls collections of permissions that may include complex operations such as an e-commerce transaction, or may be as simple as read or write. A role in RBAC can be viewed as a set of permissions." [17].

2.3.1. Discretionary Access Control

"Discretionary access control (DAC) is a policy determined by the owner of an object. The owner decides who is allowed to access the object and what privileges they have.

Two important concepts in DAC are:

- File and data ownership: Every object in the system has an owner. In most DAC systems, each object's initial owner is the subject that caused it to be created. The access policy for an object is determined by its owner.
- Access rights and permissions: These are the controls that an owner can assign to other subjects for specific resources." [17].

2.3.2. Mandatory Access Control

"Mandatory access control refers to allowing access to a resource if and only if rules exist that allows a given user to access the resource. It is difficult to manage but its use is usually justified when used to protect highly sensitive information. Examples include certain government and military information. Management is often simplified (over what can be required) if the information can be protected using hierarchical access control, or by implementing sensitivity labels. What makes the method "mandatory" is the use of either rules or sensitivity labels.

- Sensitivity labels: In such a system subjects and objects must have labels assigned to them. A subject's sensitivity label specifies its level of trust. An object's sensitivity label specifies the level of trust required for access. In order to access a given object, the subject must have a sensitivity level equal to or higher than the requested object.
- Data import and export: Controlling the import of information from other systems and export to other systems (including printers) is a critical function of these systems, which must ensure that sensitivity labels are properly maintained and implemented so that sensitive information is appropriately protected at all times.

Two methods are commonly used for applying mandatory access control:

- Rule-based (or label-based) access control.
- Lattice-based access control." [17].

2.3.3. Attribute-based access control

"In attribute-based access control (ABAC), access is granted not based on the rights of the subject associated with a user after authentication, but based on attributes of the user. The user has to prove so called claims about his attributes to the access control engine. An attribute-based access control policy specifies which claims need to be satisfied in order to grant access to an object. For instance the claim could be "older than 18". Any user that can prove this claim is granted access. Users can be anonymous as authentication and identification are not strictly required. One does however require means for proving claims anonymously. This can for instance be achieved using anonymous credentials or XACML (extensible access control markup language)." [17].

2.4. Summary

Even though RBAC is well known security pattern, still there is some disagreements on what RBAC means. That's why RBAC is open to interpretation by researchers, system developers and especially security pattern and application designers.

"Sophisticated variations of RBAC include the capability to establish relations between roles, between permissions and roles, and between users and roles. These role-role relations can enforce security policies, including separation of duties and delegation of authority. Previously, these relations would have required application software encoding; with RBAC, they can be specified once for a security domain" [11].

In RBAC, these relations can be predefined; assigning users to the roles is making it simple. Besides “without RBAC, it can also be difficult to determine what permissions have been authorized for what users” [11].

Chapter 3. Knowledge Acquisition in Automated Specification

"KAOS is a methodology for requirements engineering enabling analysts to build requirements models and to derive requirements documents from KAOS models" [10].

KAOS is a goal-oriented software requirements capturing approach in requirements engineering. It is a specific goal modeling method. It allows requirements for being calculated from goal diagrams.

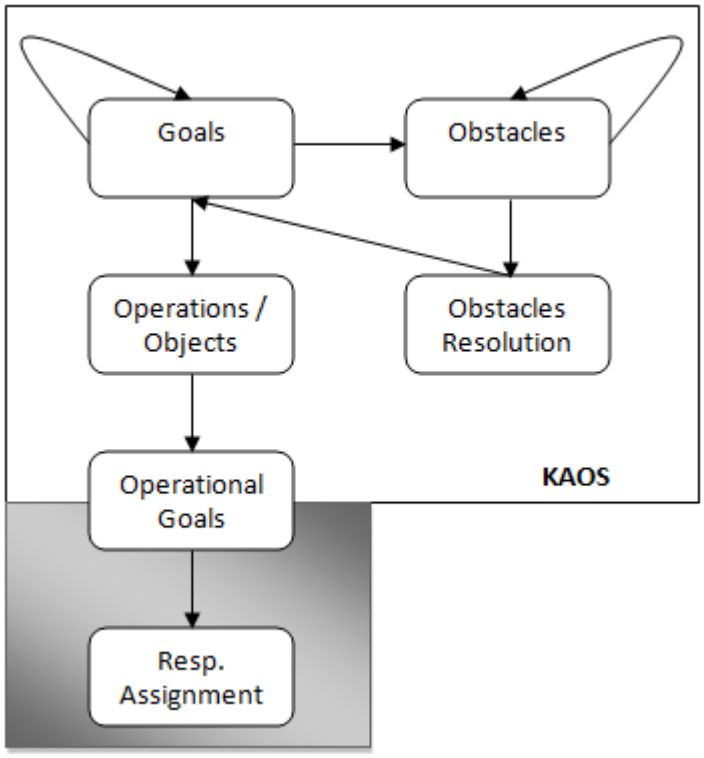

Figure 3 KAOS

(adapted from [9])

3.1. Abstract Syntax

The KAOS approach consists of a modeling language, a method, and a software environment. In this section, we will consider KAOS modeling language's abstract syntax. In other words, we present the grammar rules of KAOS.

A KAOS model includes a goal model, a responsibility model, an object model and an operation model. Each of them has a graphical and a textual syntax. We will introduce KAOS through examples from the London Ambulance Service system, and Figure 4 [4].

"A *goal* is a prescriptive assertion that captures an objective which the system-to-be should meet. Goals are either *maintain*, *avoid*, *achieve* and *cease* goals. For example, goal AccurateLocationInfoOnNonStationaryAmbulance follows the *maintain* pattern in which a property *always* holds. AmbulanceAllocationBasedOnIncidentForm follows the *achieve* pattern where a property *eventually* holds. A goal is refined through *G-refinement*, which

relates a set of subgoals whose conjunction, possibly together with domain properties, contributes to the satisfaction of the goal. A goal can have alternative G-refinements (e.g. AccurateStationaryInfo). A set of goals is conflicting if these goals cannot be achieved together (e.g. LocationContactedByPhone and InformationSentByEMail). This means that under some boundary condition these goals become logically inconsistent in a considered domain" [6].

"An *object* (e.g. Ambulance in the object model in Figure 4) is a thing of interest in the system. Its instances can be distinctly identified and may evolve from state to state. Objects have *attributes*. Goals *concern* objects and attributes (see Def in textual goal syntax in Figure 4). An *agent* plays a role towards a goal's satisfaction by *monitoring* or *controlling* object behavior. Goals are refined until they are *assigned* to individual agents. A goal effectively assigned to a *software agent* (e.g. CAD - Computer Aided Dispatch) is called a *requirement*. A goal effectively assigned to an *environment agent* (e.g. Ambulance Staff) is called an *expectation* (*assumption* in [4]). An *operation* is an *input-output* relation over objects. Operations are characterized textually by *domain* and *required* conditions. Whenever the required conditions hold, performing the operations satisfies the goal. If a goal is *operationalised* and has a responsible agent, the latter *performs* the operations (see operation model in Figure 4)" [6].

3.2. KAOS Models

3.2.1. Goal Model

"The KAOS Goal Model is the set of interrelated goal diagrams that have been put together for tackling a particular problem" [10]. A KAOS goal model is a directed graph (which is more general than a simple tree), which means that a given goal can appear on different diagrams to refine different higher-level goals. Figure 5 shows metamodel of the KAOS goal model, [4] and [15].

3.2.2. Responsibility Model

"The KAOS responsibility model is the set of derived responsibility diagrams" [10]. The responsibility model contains all the responsibility diagrams. A responsibility diagram describes for each agent, the requirements and expectations that he's responsible for, or that have been assigned to him. To build a responsibility diagram, the analyst reviews the different requirements and expectations in the goal model and assigns an agent to each of them.

3.2.3. Object Model

"The KAOS object model contains objects, agents, entities and relationships among them. The notation used in the object model complies with the one used in UML for class diagrams" [10].

The object model is used to define and document the concepts of the application domain that are relevant with respect to the known requirements and to provide static constraints on the operational system that will satisfy the requirements.

Three types of objects may coexist in the object model:

- ***Entities***: they represent independent, passive objects. 'Independent' means that their descriptions needn't refer to other objects of the model. They may have attributes

whose values define a set of states the entity can transition to. They are 'passive' means they can't perform operations.

- ***Agents***: they represent independent, active objects. They are active meaning they can perform operations. Operations usually imply state transitions on entities.
- ***Associations***: they are dependent, passive objects. 'Dependent' because their descriptions refer to other objects. They can have attributes whose values define the set of states the entity can transition to. They are passive so they can't perform operations. But agents can make association instances change state by performing operations.

The KAOS object model is compliant with UML class diagrams in that KAOS entities correspond to UML classes; and KAOS associations correspond to UML binary association links or n-ary association classes. Inheritance is available to all types of objects (including associations). Objects can be qualified with attributes.

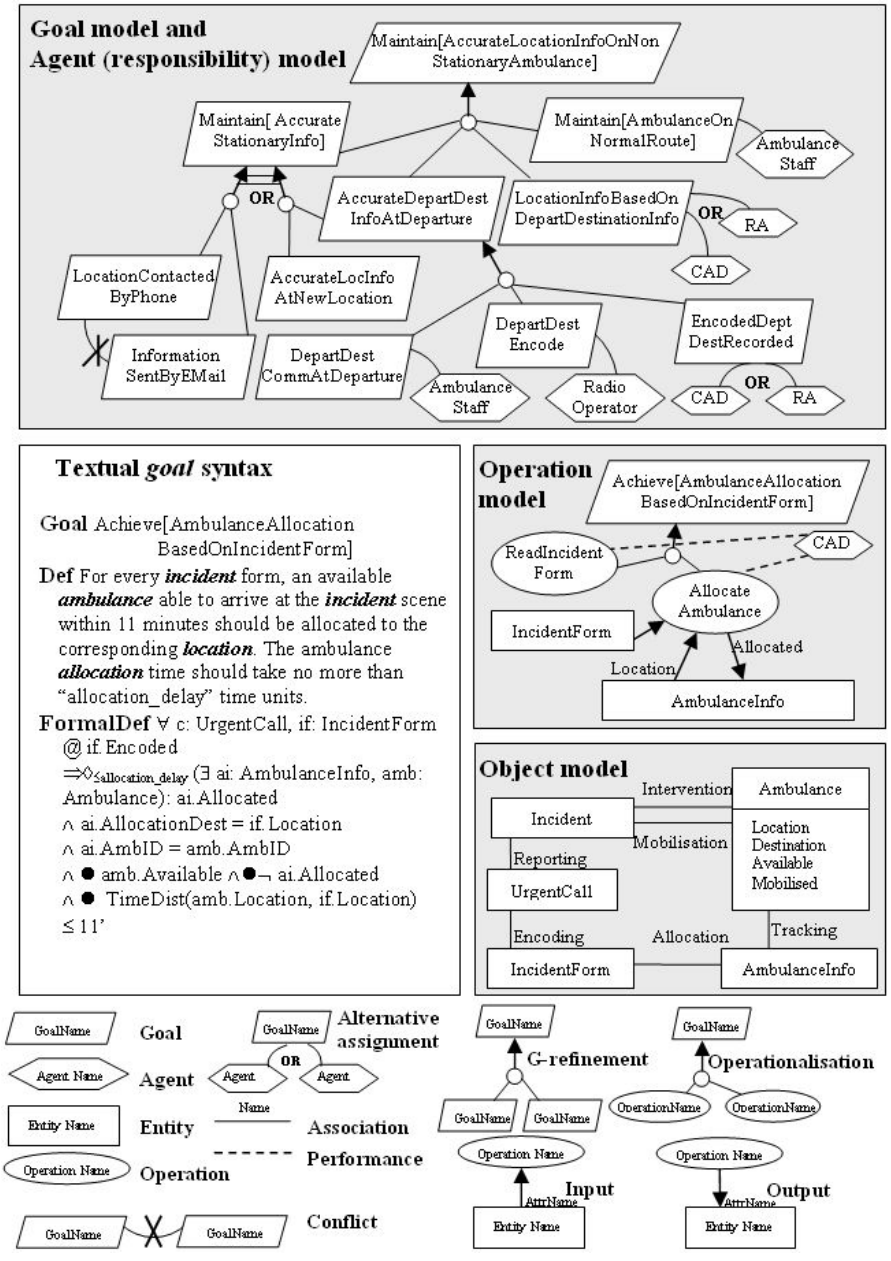

Figure 4 KAOS Model Fragment for the London Ambulance Service System

(directly taken from [4])

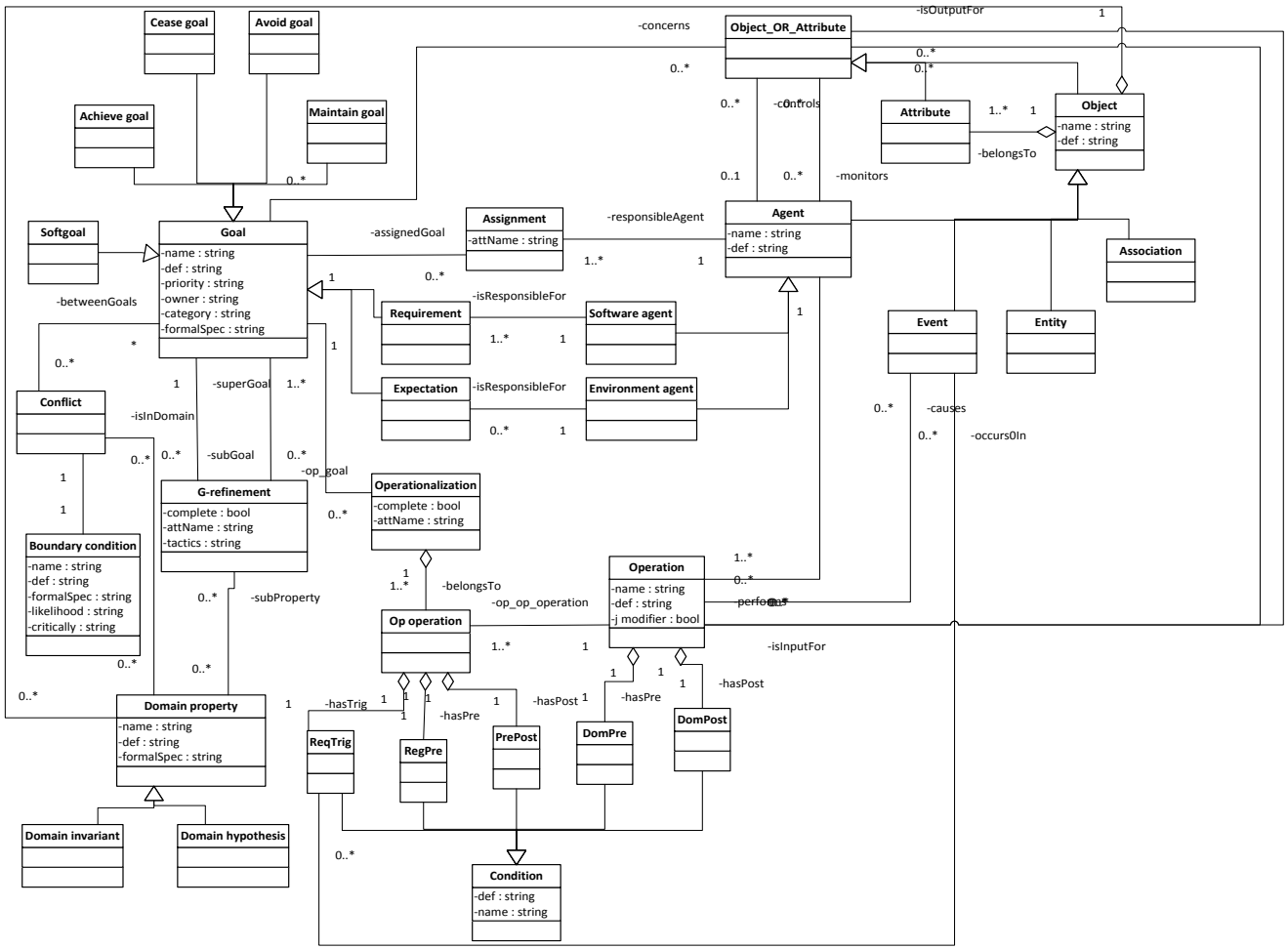

Figure 5 A Metamodel of the KAOS Goal Model

(adapted from [15])

3.2.4. Operation Model

"The KAOS operation model sums up all the behaviors that agents need to have to fulfill their requirements. Behaviors are expressed in terms of operations performed by agents. Those operations work on objects described in the object model: they can create objects, provoke object state transitions or trigger other operations through sent and received events" [10].

The KAOS operation model describes all the behaviors that agents need to fulfill their requirements. Behaviors are expressed in terms of operations performed by agents. Operations work on objects, they can create objects, trigger object state transitions and activate other operations.

A KAOS operation diagram typically composes operations performed by one or several agents to achieve a requirement. Compositions are made through data flows (the output of an operation output becomes the input of another operation) or control flow (an event sent by an operation triggers or stops another operation). An operation diagram thus describes how the agents need to cooperate in order to make the system work. With KAOS, the operation model is connected to the goal model: the analysts justify operations by the goals they "operationalize". An operation with no justification means that either there is still missing goals in the model or that the operation is not necessary. Conversely if some requirements are left without "operationalization", they may just be wishful thinking.

3.3. Concrete Syntax

Besides abstract syntax, KAOS also has concrete syntax like spoken languages do. In spoken languages, there are letters and words but in KAOS, there are constructs.

"Agent: Active Object (=processor) performing operations to achieve goals. Agents can be the software being considered as a whole or parts of it. Agents can also come from the environment of the software being studied; human agents are in the environment.
Association: Object, the definition of which relies on other objects linked by the association.
Composite system: The software being studied and its environment.
Conflict: Goals are conflicting if under some boundary condition the goals cannot be achieved altogether.
Domain Property: Descriptive assertion about objects in the environment of the software. It may be a domain invariant or a hypothesis. A domain invariant is a property known to hold in every state of some domain object, e.g., a physical law, regulation, … A hypothesis is a property about some domain object supposed to hold.
Entity: Autonomous object, that is, the definition of which does not rely on other objects.
Environment: Part of the universe capable of interaction with the software being studied.
Event: Instantaneous object (that is, an object alive in one state only) which triggers operations performed by agents.
Expectation: Goal assigned to an agent in the environment.
Formal model: Model in which the concepts have been mathematically formalized.
Goal: Prescriptive assertion capturing some objective to be met by cooperation of agents; it prescribes a set of desired behaviors. Requirements and expectations are goals.
Model: Abstract representation of a composite system. An model represents a composite system by means of concepts of different types, mainly, objects, desired or undesired properties (goals, obstacles), and behaviors (operations).

Object: Thing of interest in the composite system being modeled whose instances can be distinctly identified and may evolve from state to state. Agents, events, entities and associations are objects.
Obstacle: Condition (other than a goal) whose satisfaction may prevent some goal(s) from being achieved; it defines a set of undesired behaviors.
Operation: Specifies state transitions of objects that are input and/or output of the operation. Operations are performed by agents.
Operationalisation: Relationship linking a requirement to operations. Holds when each execution of the operations (possibly constrained to that intent) will entail the requirement. Makes the connection between expected properties (goals) and behaviors (operations).
Refinement: Relationship linking a goal to other goals that are called its subgoals. Each subgoal contributes to the satisfaction of the goal it refines. The conjunction of all the subgoals must be a sufficient condition entailing the goal they refine.
Requirement: Goal assigned to an agent of the software being studied.
Responsibility: Relationship between an agent and a requirement. Holds when an agent is assigned the responsibility of achieving the linked requirement.
Semi-formal model: Model in which the concepts are not mathematically formalized, every concept in the model receives a name, a type, a textual definition, values for attributes and a graphical representation." [10].

3.4. Semantics

"The KAOS approach provides support for security goal specification in terms of a number of specialized meta-classes of goal, namely, Confidentiality, Integrity, Availability, Privacy, Authentication and Non-repudiation goal subclasses. In order to support the concepts of attacker knowledge, the formal language of goals is extended with the epistemic operators, KnowsVag, which is defined as follows:

$KnowsV_{ag}$ (v) = ∃x: $Knows_{ag}$(x=v) ("knows value")
$Knows_{ag}$(P) = $Belief_{ag}$ (P) ∧ P ("knows property)

The operational semantics of the epistemic operator $Belief_{ag}$(P) is defined as "P being one of the properties stored in the local memory of agent ag". The knowledge of a value of a property at a given point depends on both the agent having a value for the property in its local memory and that property value actually holding at the given point in time.

The use of obstacles for security goals makes obstacle refinement trees analogous to the threat trees that are used for modeling potential attacks security-critical systems. However, obstacles neither capture the goals and knowledge of a potential attacker; or the vulnerabilities in software systems. The notions of anti-goals and anti-models were introduced to the KAOS framework in order to deal with these problems. Combining with the epistemic operators described above, allows security patterns to be expressed in the KAOS framework" [9].

3.5. KAOS Extension to Security

KAOS is based on goals and these goals are operationalized into specifications of operations to achieve them. Besides that, goals refer to objects which can be derived from their specification to create UML class diagrams as a structural model of a system. Along this process, because of obstacles sometimes it is needed to generate some alternative resolutions such as: "goal substitution, agent substitution, goal weakening, goal restoration, obstacle prevention and obstacle mitigation" [13]. Obstacles are a means for identifying

goal violation scenarios. In declarative terms, an obstacle to some goal is a condition whose satisfaction may prevent the goal from being achieved. "Richer models should thus be built to capture attackers, their goals and capabilities, the software vulnerabilities they can monitor or control, and attacks that satisfy their goals based on their capabilities and on the system's vulnerabilities" [13].

Anti-goals are the goals of attackers which includes malicious obstacles to security goals. Anti-goals should be distinguished from the goals the system under consideration should satisfy. Anti-model is a model that exhibits how specifications of model elements could be maliciously threatened, why and by whom. "Anti-models should lead to the generation of more subtle threats and the derivation of more robust security requirements as anticipated countermeasures to such threats" [13]. Table 2 shows the anti-model building method [13].

Table 2 Anti-Model Building Method

(adapted from [13])

1. Get initial anti-goals by negating relevant Confidentiality, Privacy, Integrity and Availability goal specification patterns instantiated to sensitive objects from the object model.
2. For each such anti-goal, elicit potential attacker agents that might own the anti-goal, from questions such as "*WHO can benefit from this anti-goal?*" (Applicationspecific specializations of known attacker taxonomies may help answering such questions).
3. For each anti-goal and corresponding attacker class(es) identified, elicit the attacker's higher-level anti-goals from questions such as "*WHY would instances of this attacker class want to achieve this anti-goal?*". Such questions may be asked recursively to elicit more and more abstract anti-goals yielding threat rationales together with other potential threats from alternative refinements of those higher-level anti-goals.
4. Elaborate the anti-goal AND/OR graph by AND refining/abstracting anti-goals along alternative branches, with the aim of deriving terminal anti-goals that are realizable either by the identified attacker agents or by attackee software agents. The former are anti-requirements assigned to the attacker whereas the latter are vulnerabilities assigned to the attackee.
5. Derive the object and agent anti-models from anti-goal specifications. The boundary between the anti-machine (under the attacker's control) and the anti-environment (which includes the software attackee) are thereby derived together with monitoring/control interfaces.
6. AND/OR-operationalize all anti-requirements in terms of potential capabilities of the corresponding attacker agent – the latter may include blind or intelligent searching, eavesdropping, deciphering, spoofing, cookie installation, etc.

3.6. Summary

A lot of requirements documents produced nowadays just describe solutions: the expected functions, processes and data structures. However it should become clear to the reader that a requirements analysis with KAOS is much more than a limited description of the solution. An important focus is put on the problem itself. If we compare the kind of information provided by the solution description with the one provided by the problem

description, we will see that the information collected in the latter diagrams are not irrelevant for the requirements document. They introduce abstract and fundamental properties that have to be fulfilled by the system to be. If a requirements document consisting only of description derived from solution description, one can reasonably expect that a development team will develop the system right with respect to that specification. But how may we guarantee that the system built is the right system if we discard the first part of the analysis which describes precisely what the users really need?

Some of the benefits of KAOS are listed below [10]:

- "*Traceability*: A major benefit of KAOS resides in the fact that it provides continuum between the problem description and the expected solution description. This bi-directional traceability between problem and solution spaces is fundamental not only for the requirements analyst to be sure, the system to build will be the right one, but also for developers who need to understand the context and objectives to make correct architectural and design choices. Moreover systems developed nowadays work in a quickly changing environment that requires lots of modifications. As with KAOS, the requirements document is derived from a KAOS model, it becomes possible to modify the KAOS model and regenerate a consistent requirements document from it.

- *Completeness*: Requirements documents elaborated with KAOS tend to be more complete. A complete KAOS model leaves no space for wishful thinking (a goal not refined), no space for requirements for which we do not know who is responsible for, no space for unjustified operations, and no space for operations, for which we ignore who will execute what and when. Completeness of a KAOS model clearly relies also on the completeness of the goal model.

- *No ambiguity*: On the one hand, the completeness criteria contribute to less ambiguity in requirements documents; we know who is responsible for what and who perform what. On the other hand, the object model contains all the information needed to produce the requirements document glossary. The glossary validation forces all stakeholders who generally have different background, to agree on the domain and application relevant concepts. Standards for requirements document require the inclusion of a glossary. With KAOS, we can build the glossary progressively and we get for free a criterion for deciding which concept has to be defined in the glossary: in fact all those defined in the object model."

Chapter 4. Related Work

Security remains a key challenge in the development of software systems and the goal of developing secure software systems has remained an area of active research. Research in security engineering has resulted in the realization that documenting recurring security problems and their solutions as security patterns is an important advancement as it allows software designers with little knowledge of security to build secure systems. When a designer encounters a security problem that match a given pattern, they can reuse the solution part of the pattern or use the pattern to guide them in finding a solution to the problem at hand. In this chapter we have reviewed approaches to security analysis according to RBAC. Our review focused on evaluating the capabilities of these approaches to supporting security analysis patterns and is based on a set of evaluation criteria for characterizing security patterns.

4.1. SecureUML for RBAC

"Lodderstedt *et al*. [5] present a modeling language, based on UML, called SecureUML. SecureUML focuses on modeling access control policies and how these policies can be integrated into a model-driven software development process. It is based on an extended model of role-based access control (RBAC) and uses RBAC as a meta-model for specifying and enforcing security. RBAC lacks support for expressing access control conditions that refer to the state of a system, such as the state of a protected resource. In addressing this limitation, SecureUML introduces the concept of authorization constraints. Authorization constraints are preconditions for granting access to an operation."

The SecureUML meta-model based on the RBAC model is shown in Figure 6 [5]. It describes the abstract syntax with UML diagrams and its information about access control. The meta-model shows concepts (User, Role, and Permission), UML elements (ModelElement), and permissions/constraints. The combination of the graphical capability of UML, access control properties of RBAC, and authorization constraints makes it possible to base access decision on dynamically changing data such as time. Similar to its parent modeling language UML, SecureUML focuses on the design phase of software development.

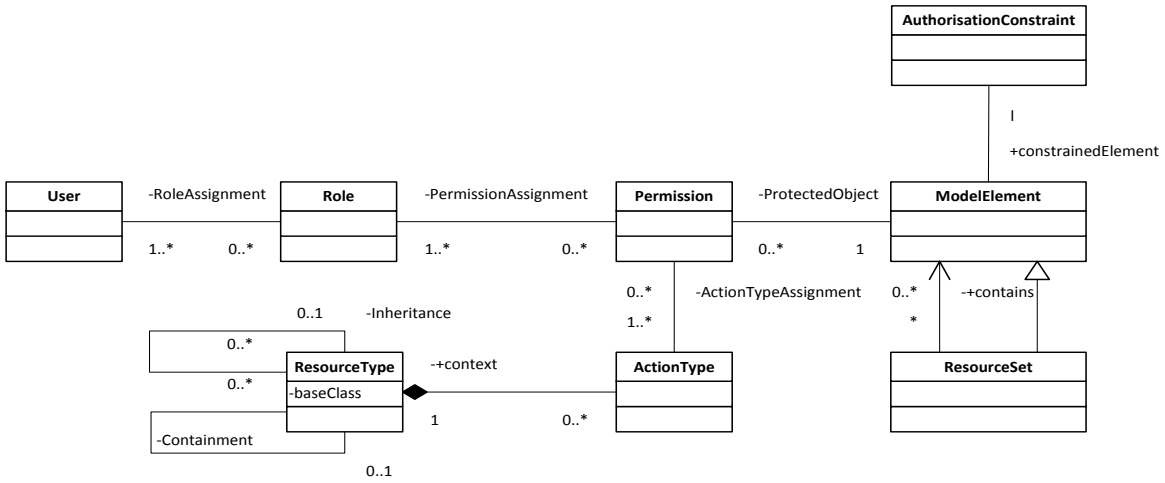

Figure 6 SecureUML Meta-model

(adapted from [5])

4.2. UMLsec for RBAC

"UMLsec (Jurjens, 2004) is an extension of UML which allows an application developer to embed security-related functionality into a system design and perform security analysis on a model of the system to verify that it satisfies particular security requirements. Security requirements are expressed as constraints on the behavior of the system and the design of the system may be specified either in a UML specification or annotated in source code" [2].

UMLsec is defined as a UML profile extension using stereotypes, tags and constraints. Role-based access control stereotype, <<rbac>>, its tagged values and constraints is a subset of UMLsec. <<rbac>> stereotype enforces RBAC in the business process specified in the activity diagram.

Table 3 UMLsec Rbac Stereotype

(adapted from [2])

Stereotypes	Base class	Tags	Constraints	Description
Rbac	subsystem	protected, role, right	only permitted activities executed	enforces RBAC

"The UMLsec approach consists of two main steps. The first step is translating UML models into UMLsec specifications. UMLsec specifications describe the behavior of a system in terms of its components and their interaction. The behavior of system components is described in terms of the messages they exchange in communication links between them. The next step, security analysis, involves eliciting ways by which an adversary may modify the contents of the data exchanged in communication link queues that may compromise the integrity of system behavior. The analysis focuses on a consideration specific types of adversaries that may attack a system in a specific way. An example of such an attack on a communication link between components is breach of confidentiality, which state that some information will only become known only to legitimate parties. UMLsec specifications are checked for vulnerability to types of threats on contents of a communication link such as delete, read, and insert. The types of threats are adversary actions associated with particular adversary types. *Delete* means that an adversary may delete messages from a communication link queue. *Read* allows an adversary to read messages in the link queue, while *insert* allows the adversary to insert messages in the communication link" [9].

4.3. Summary

Table 4 shows the general comparison of SecureUML, UMLsec and KAOS based on *problem*, *context*, *forces*, *solution*, and *consequences* criteria. These criteria are chosen based on the research done by A.Nhlabatsi, A.Bandara in 2009 [9].

Table 4 General Comparison of SecureUML, UMLsec, and KAOS

(adapted from [9])

	SecureUML	UMLsec	KAOS
Problem	SecureUML does not explicitly model security goals but focuses on modeling solutions to security problems. Its foundation of on RBAC implies that it is specific to security goals relating to controlling access to shared resources.	Although security analysis is guided by specific goals and constraints in checking for security vulnerabilities in a system design, UMLsec does not have a specific construct for modeling security problems.	The intent of a security requirements pattern expressed in KAOS is documented in the top-level goal of the pattern. The meta-class of the top-level goal will identify if the pattern pertains to a confidentiality, integrity, availability, privacy, non repudiation or authentication concern. The anti-goal model that forms part of the pattern definition can be used to identify the problem addressed by the pattern.
Context	The modeling of context in SecureUML is similar to RBAC. However, the context only captures assets that may be harmed in the event of an attack. It does not model scenarios of attacks and possible harm to assets.	Yes, the UMLsec approach explicitly models context of a security problem. However this context is limited to system design components, their interactions, and adversary models.	As with the intent, the general context of the problem the pattern aims to address will be documented in the top-level anti-goal. More specific details of the attacker knowledge, intention and asset properties will be captured in lower level goals of the pattern definition. The notation does not provide an explicit means of specifying harms to assets, although these can be captured as annotations to the anti-goal model.
Forces	There is no construct for capturing and modeling forces in SecureUML.	Once security vulnerabilities have been identified the system design is progressively refined to eliminate the threat. The rationale for selecting a	The KAOS pattern notation does not provide an explicit means of capturing the forces that might influence the selection of a particular refinement

		particular solution of refining a design is not explicitly captured and it is not explicit whether alternative solutions are explored. It is possible though that such alternative security solutions can be explored in the refinement process based on the native UML design.	pattern. However, requirements engineers are able to use the preconditions specified in the formal definition of goals to determine the suitability of a give pattern for the problem at hand.
Solution	Yes. The combination of RBAC with UML and the authorization constraints extension is the bases of a security solution in SecureUML.	UMLsec provides an explicit refinement of design in order to ensure that they satisfy security constraints. Once a design has undergone refinement its ability to satisfy security requirements is re-verified. The refinement continue until it can be demonstrated that the vulnerability of the design to attacks is eliminated	The KAOS pattern notation allows specification of the solution to the initial problem in the form of sub-goals that satisfy the original goal.
Consequences	Yes. The consequences of using SecureUML is a solution to an access control problem in access rights to resource are assigned to roles and users are assigned to roles with specific authorization constraints.	When a design has been found to violate security requirements, UMLsec provides for the generation of scenarios, in the form of attack sequences, which explain how security requirements may be violated by the design. The results (consequences) of refining a system design in order to address security vulnerabilities are captured in the revised version of the design and assessed against security requirements.	The consequence of a KAOS refinement pattern is to satisfy the original, high-level goal. If a pattern is specified using the formal notation provided by KAOS, the entailment relation between the sub-goals and top-level goal can be formally proven. This ability to validate that the consequences specified for a given pattern are correct is particularly useful in the domain of security patterns.

PART II

CONTRIBUTION

In Contribution Part, we are going to present the common concepts and similarities between KAOS and RBAC. Later on, we will generate transformation rules between the security modeling approaches (KAOS-SecureUML, SecureUML-KAOS, KAOS-UMLSec, and UMLSec-KAOS). We will apply these transformation rules on to the models, KAOS, SecureUML and UMLSec separately then we will get transformed versions of these models.

Chapter 5. Understand the Language

In this chapter, we will try to show the common concepts and similarities between KAOS and RBAC. In order to do this, first of all, we will explain the common constructs between them and secondly make a design for transformation rules based on these common constructs.

5.1. Common Concepts between KAOS and RBAC

In this section, we explain the common constructs between KAOS and RBAC. We will show what their definitions are and how they match to each other.

5.1.1. Entity - Object

In RBAC, "Object is defined as a passive entity that contains or receives information" [11]. In KAOS, "Object is a thing of interest in the software being studied and its environment, being modeled whose instances can be distinctly identified and may evolve from state to state. Agents, events, entities and associations are objects" [10]. From these definitions, we understand that they both refer to similar concepts, changeable status or value. There is a slight difference about objects in RBAC and KAOS. In RBAC, objects are passive entities which mean they cannot perform operations. On the other hand, in KAOS, objects can be either passive or active which can perform operations (e.g. agents) and which cannot (e.g. entities and associations). So we can match objects in RBAC as in entities in KAOS.

5.1.2. Agent - Role

In RBAC, "Role means a job function within the organization that describes the authority and responsibility conferred on a user assigned to the role." [11]. In KAOS, there is no construct named as role but there are agents which refers "active objects performing operations to achieve goals." [10]. We understand more clearly why agent matches with role with the help of sub divisions of agents. "Agents can be the software being considered as a whole or parts of it. Agents can also come from the environment of the software being studied; human agents are in the environment." [10]. This means, there are two types of agents: Software agents and environment agents. Both of them are active components that play some role towards goal satisfaction.

5.1.3. Operation - Operation

In RBAC, "An operation is an executable image of a program, which upon invocation executes some function for the user." [1]. In KAOS, operation is expressed as "an input-output relation over objects; operation applications define state transitions." [14]. Both definitions refer to the same thing, there is an execution and these operations are characterized by pre and post conditions.

5.1.4. Performance Links - Permission Assignment

In RBAC, "permission assignment is an authorized interaction a subject can have with an object." [11]. It is between roles and permissions (operations and objects) with possible constraints. In KAOS, there are operations and there are agents who should perform these operations. In order to determine which agent has *permission* to perform which particular operation, performance links are used.

5.1.5. User Entity - User

In RBAC, user is defined as “any person who interacts directly with a computer system.” [11]. Basically, user is a human-being. In KAOS, there is no such a construct that we can use as user. Therefore, we have to find an alternative way to represent user in KAOS. The solution is to create an entity named user which leads us to create human objects of it.

5.2. Design of Transformation Rules

In this section, we generate a transformation process that leads us to receive transformed model as it is shown in Fig. 7. “Input” represents the initial model, “Action” represents the transformation rules applied to the input and “Output” is the outcome transformed model.

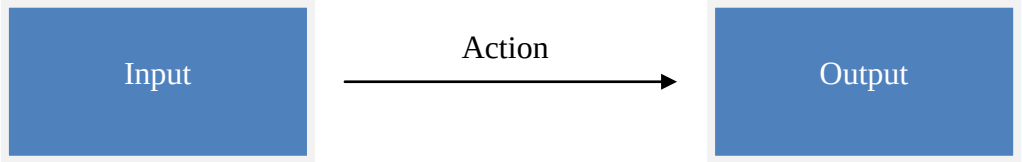

Figure 7 Transformation Process

5.3. Summary

In this chapter, we showed the similarities between KAOS and RBAC regarding their constructs. We first gave the definitions of these constructs and later on we explained which construct of KAOS match with which construct of RBAC. This is very important stage in our research because for the following stages, we will rely on these matches to prove our transformation rules. More detailed examples will be shown in transformation rules parts in Chapter 6 and Chapter 7. In Fig. 8, we see extract of the KAOS metamodel showing which elements are part of RBAC.

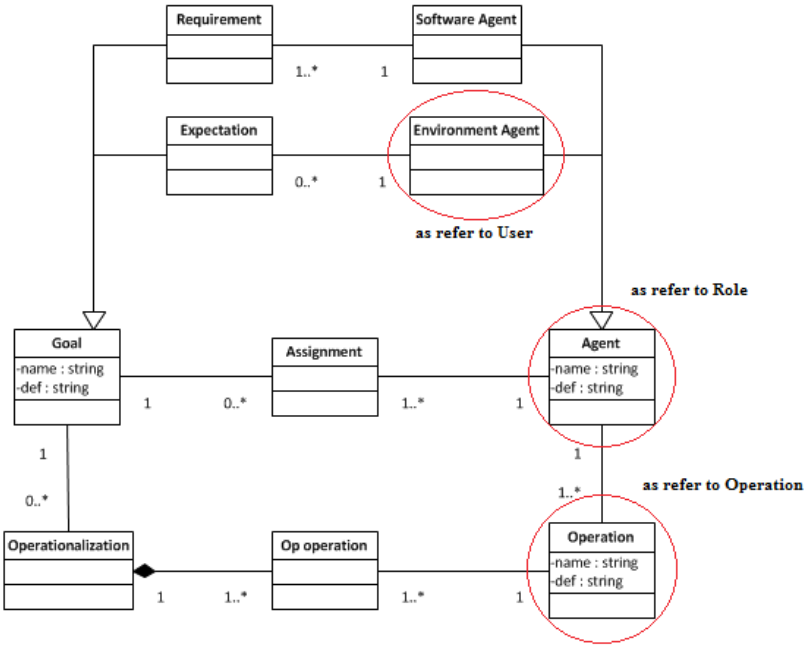

Figure 8 Extract of KAOS Metamodel

Chapter 6. SecureUML - KAOS Transformations

In this chapter, we describe a scenario called Meeting Scheduler Example. According to this scenario, we create SecureUML diagram and later on we generate transformation rules from SecureUML to KAOS and vice versa.

SecureUML inherits features from RBAC and UML. It is based on these concepts: Role, Permission, ResourceSet, ModelElement, ActionType, and AuthorizationConstraints where ModelElement is a UML concept and the others are RBAC concepts.

6.1. Meeting Scheduler Example with SecureUML

The Meeting scheduler example is described as follows: "Meeting initiator needs to organize a top-secret meeting. He needs to invite potential Meeting participants and find a suitable meeting place and time. In order to ease his task Meeting initiator decides to use a Meeting scheduler system for sending invitations, merging availability dates and informing the Meeting participants. Since the Meeting is top secret, the Meeting scheduler system must apply appropriate security policy for the Meeting agreement (place and time). This means, the time and place could be entered and changed only by the Meeting initiator and could be viewed only by the invited Meeting participants. In other words, no unintended audience should get access to the Meeting agreement." [16].

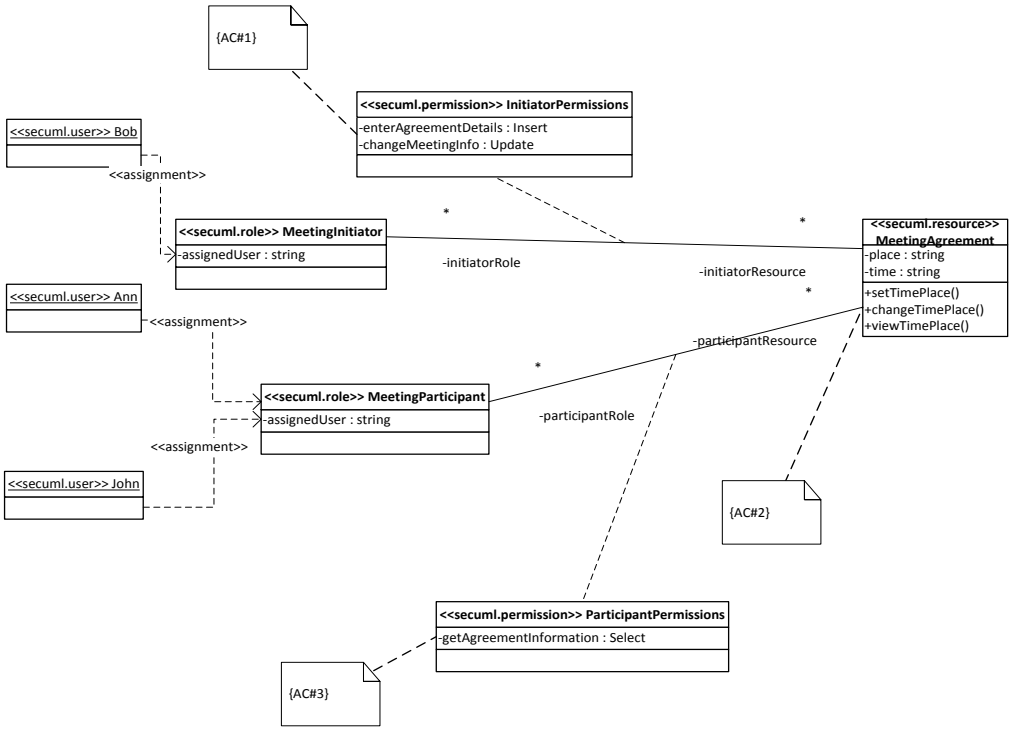

Figure 9 SecureUML Diagram for Meeting Scheduler Example

(adapted from [7])

There are three authorization constraints: AC#1, AC#2 and AC#3 to strengthen the permissions.

AC#1
```
context MeetingAgreement::setTimePlace():void
pre: self.roleInitiator.assignedUser ->
        exists(i | i.assignedUser = “Bob”)
```

AC#2
```
context MeetingAgreement::changeTimePlace():void
pre: self.roleInitiator.assignedUser ->
        exists(i|i.assignedUser = “Bob”)
```

AC#3
```
context MeetingAgreement::viewTimePlace():void
pre: self.roleParticipant->
        exists (p1|p1.assignedUser="Ann")and
    self.roleParticipant->
        exists (p2|p2.assignedUser="John")and
    self.roleParticipant->size = 2
```

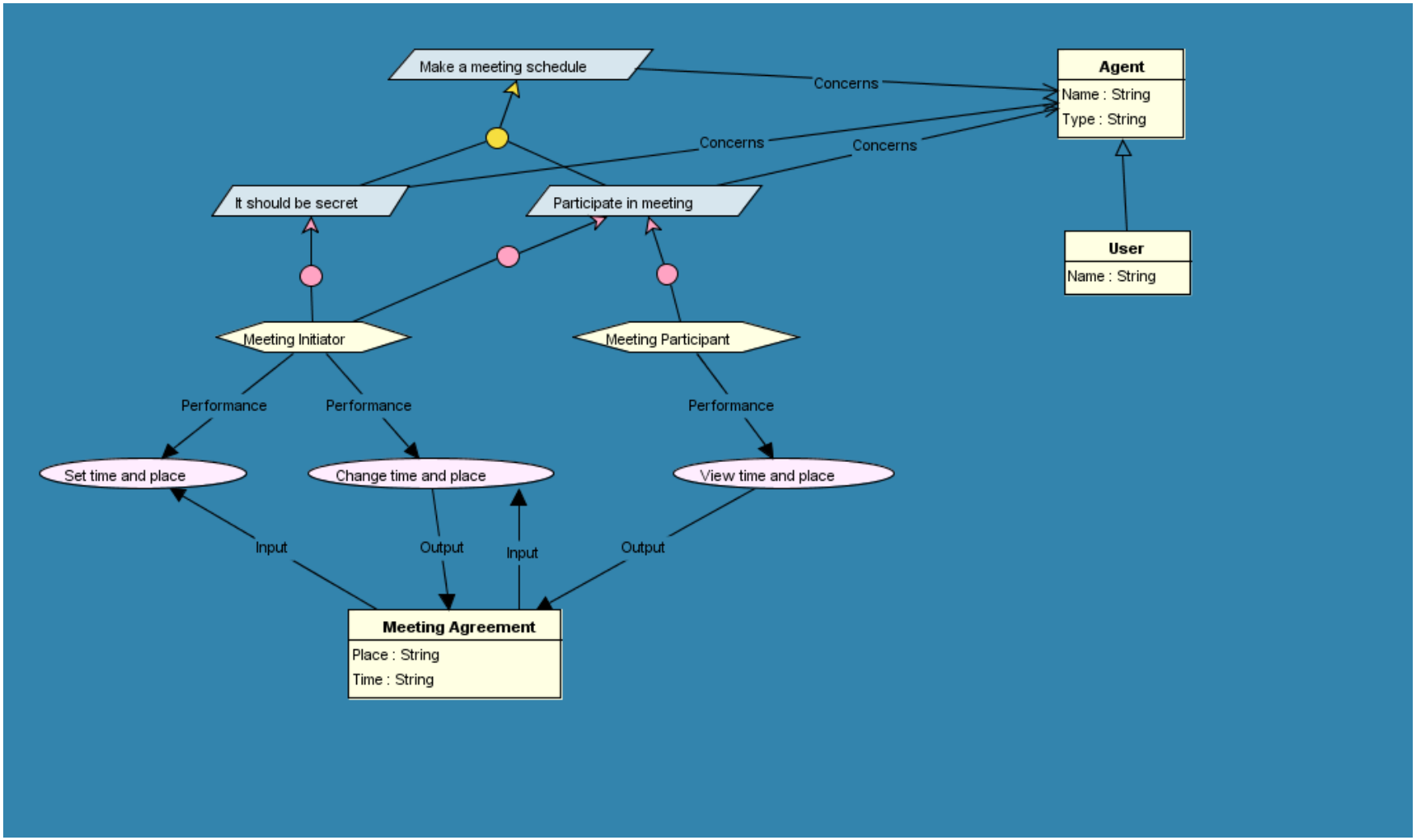

Figure 10 Meeting Scheduler Example with KAOS

6.2. Transformation Rules

There are two sets of transformation rules, first it is from SecureUML to KAOS and the other one is from KAOS to SecureUML.

6.2.1. Model Transformation from SecureUML to KAOS

We will use Fig. 9 SecureUML diagram for meeting scheduler example as our *input*. Below we define four transformation rules to transform a model from SecureUML to KAOS, these are our actions and the final figure that we have Fig. 14 SecureUML to KAOS Transformation Rule # 4 will be our output.

SK1. A SecureUML class with the stereotype <<secuml.role>> is transformed to the agents.
Example: Roles become agents. The names of them remain the same (Meeting Initiator, Meeting Participant).

Figure 11 SecureUML to KAOS Transformation Rule # 1

SK2. The SecureUML association class with the stereotype <<secuml.permission>> becomes performance links between agents and corresponding attributes in KAOS model.
Example: In KAOS permission assignments are handled by performance links. It shows which agent has permission to do which operation. E.g. Meeting Initiator is performing change time and place.

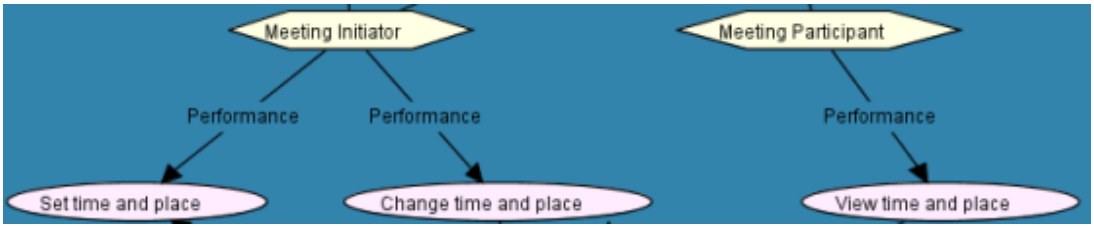

Figure 12 SecureUML to KAOS Transformation Rule # 2

SK3. A class with a stereotype <<secuml.resource>> is transformed to entities (Meeting Agreement), and the operations of this class become operations belonging to operation model in the KAOS model.
Example: the class *MeetingAgreement* (see Figure 9) is represented as an entity called Meeting Agreement in Figure 10. The operations *setTimePlace()*, *changeTimePlace()*, and *viewTimePlace()* are shown as operations respectively set time and place, change time and place, and view time and place in operation model in KAOS.

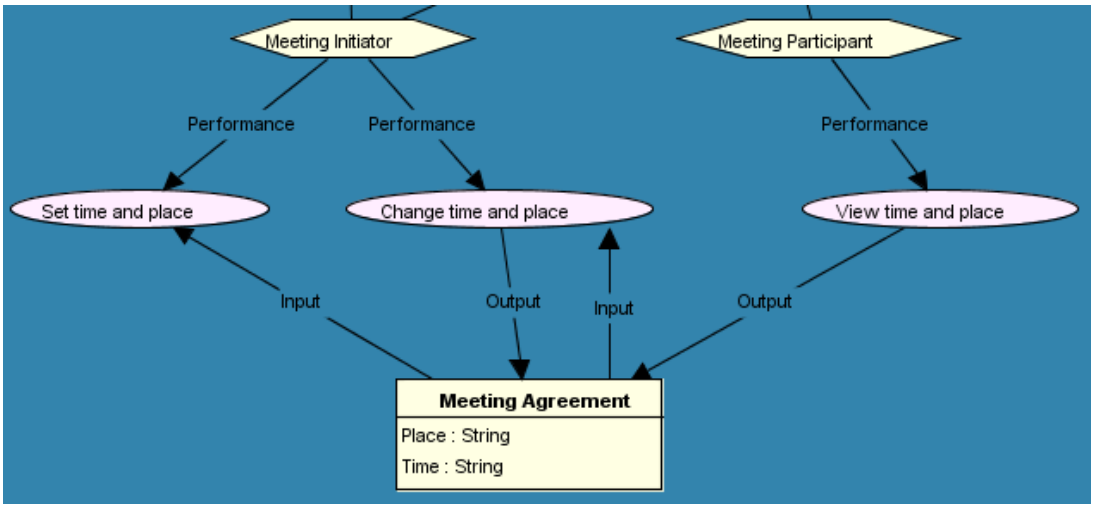

Figure 13 SecureUML to KAOS Transformation Rule # 3

SK4. A relationship with a stereotype <<assignment>> relationship used to connect users and their roles is transformed to generalization among the agents and users in KAOS model.
Example: From Figure 10 we specify "User" entity to create another object for users, as provided in Table 7, e.g. Agent → User → Bob.

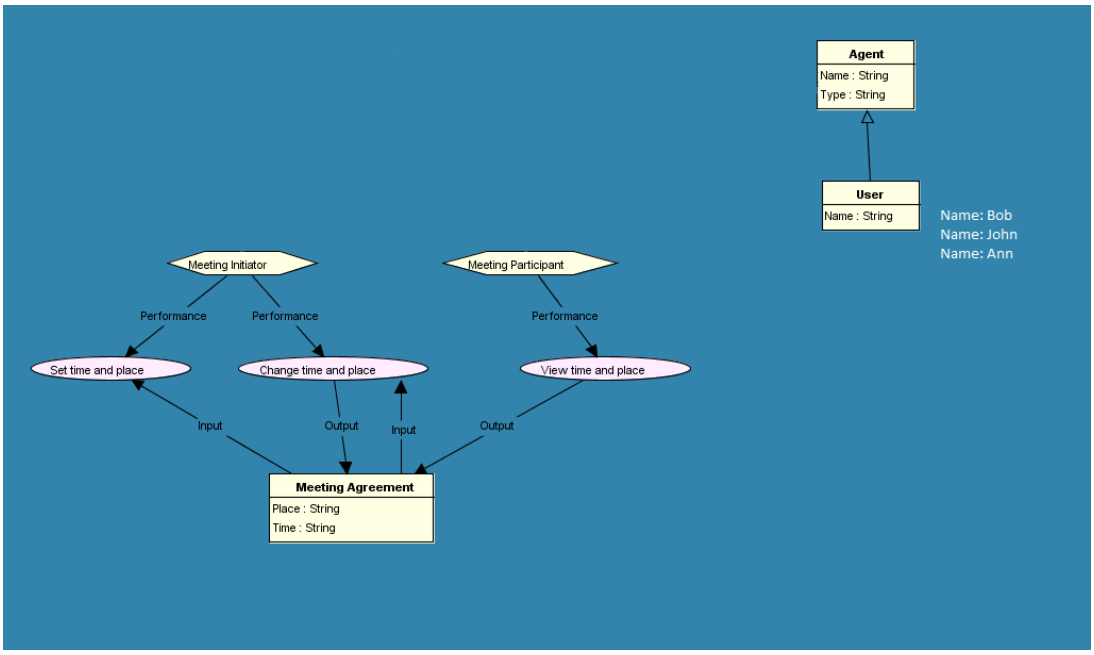

Figure 14 SecureUML to KAOS Transformation Rule # 4

Note1: From SecureUML model, we cannot directly generate the goals that we have to elicit. So, we have to focus the operations and permissions to understand the aim and then we may write the goals with some lost information.

Note2: The KAOS object model is compliant with UML class diagrams therefore KAOS entities correspond to UML classes in SecureUML; and KAOS associations correspond to UML binary association links or n-ary association classes. On the other hand in KAOS model there are few more association link types, such as *concern, input, output* etc. The

developer should analyze the relationship between the constructs and decide to link one to another. Table 5 shows the entire link types of KAOS model.

Table 5 KAOS Model Link Types

Link Type	Direction
Concerns	Link one of Goal, Softgoal, Requirement, Expectation to one of Entity, Agent, Event, N-ary Association
Performance	Link one of Agent to one of Operation
Operationalization	Link one of Operation to one of Requirement, Expectation
Responsibility	Link one of Agent to one of Requirement, Expectation
Assignment	Link one of Agent to one of Requirement, Expectation, Goal, SoftGoal
Refinement	Link one of Requirement, Expectation, Goal, SoftGoal, DomProp to one of Requirement, Expectation, Goal, SoftGoal
Resolution	Link one of Requirement, Expectation, Goal, SoftGoal to one of Obstacle
Obstruction	Link one of Obstacle to one of Requirement, Expectation, Goal, SoftGoal
Conflict	Link one of Requirement, Expectation, Goal, SoftGoal, DomProp to one of Requirement, Expectation, Goal, SoftGoal, DomProp
O_Refinement	Link one of DomProp, Obstacle to one of Obstacle
IsA	Link one of Entity, Agent, Event, N-ary Association to one of Entity, Agent, Event, N-ary Association
Binary Association	Link one of Entity, Agent, Event to one of Entity, Agent, Event
Link	Link one of N-ary Association to one of Entity, Agent, Event, N-ary Association
Monitoring	Link one of Agent to one of Entity, Agent, Event, N-ary Association
Control	Link one of Agent to one of Entity, Agent, Event, N-ary Association
Cause	Link one of Event to one of Operation
Input	Link one of Entity, Agent, Event, N-ary Association to one of Operation
Output	Link one of Operation to one of Entity, Agent, Event, N-ary Association

6.2.2. Model Transformation from KAOS to SecureUML

We will use Fig. 10 meeting scheduler example with KAOS as our *input*. Below we define four transformation rules to transform a model from KAOS to SecureUML, these are our actions and the final figure that we have Fig. 20 KAOS to SecureUML Transformation Rule # 4 will be our output.

KS1. In KAOS model, entities (independent, passive objects) are represented by stereotype <<secuml.resource>> in the SecureUML and the operations defined in KAOS model are transformed which hold these operations to the SecureUML class with a stereotype <<secuml.resource>>.
Example: Meeting Agreement entity becomes Meeting Agreement class. Attributes remain the same but additionally operations from KAOS model are also taken inside the class.

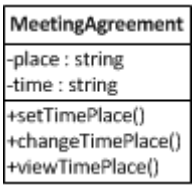

Figure 15 KAOS to SecureUML Transformation Rule # 1

KS2. In KAOS model, agents (independent, active objects) can be transformed to the <<secuml.role>> classes in SecureUML model and each of this class should have default attribute as "assignedUser : string".
Example: Agents become roles in SecureUML. The names of them remain the same. (Meeting Initiator, Meeting Participant).

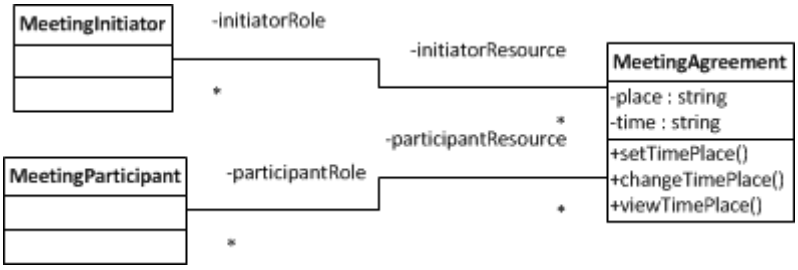

Figure 16 KAOS to SecureUML Transformation Rule # 2

KS3. In KAOS model, users are environment agents, they are derived from agent entity. They can be defined with a stereotype <<secuml.user>> in SecureUML model.
Example: Environment and software agents become users in SecureUML. E.g. Bob, Ann and John.

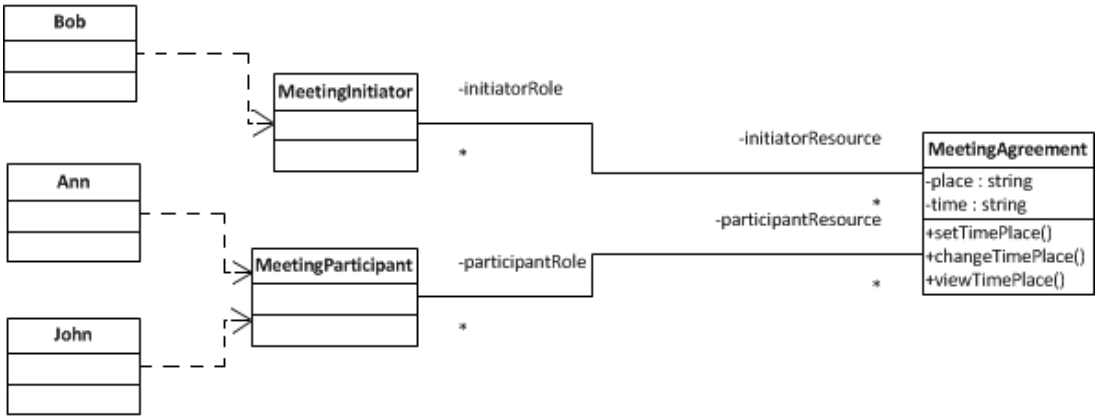

Figure 17 KAOS to SecureUML Transformation Rule # 3

KS4. In KAOS model, from performance links between agents and operations, we are able to identify on which operations a role can perform security actions. Thus, from each occurrence of this links in the KAOS model, a corresponding association class between a << secuml.roles>> and a << secuml.resource>> is introduced in SecureUML.
Example: Permission classes are introduced here to replace performance links between agents and operations. E.g InitiatorPermissions and ParticipantPermissions.

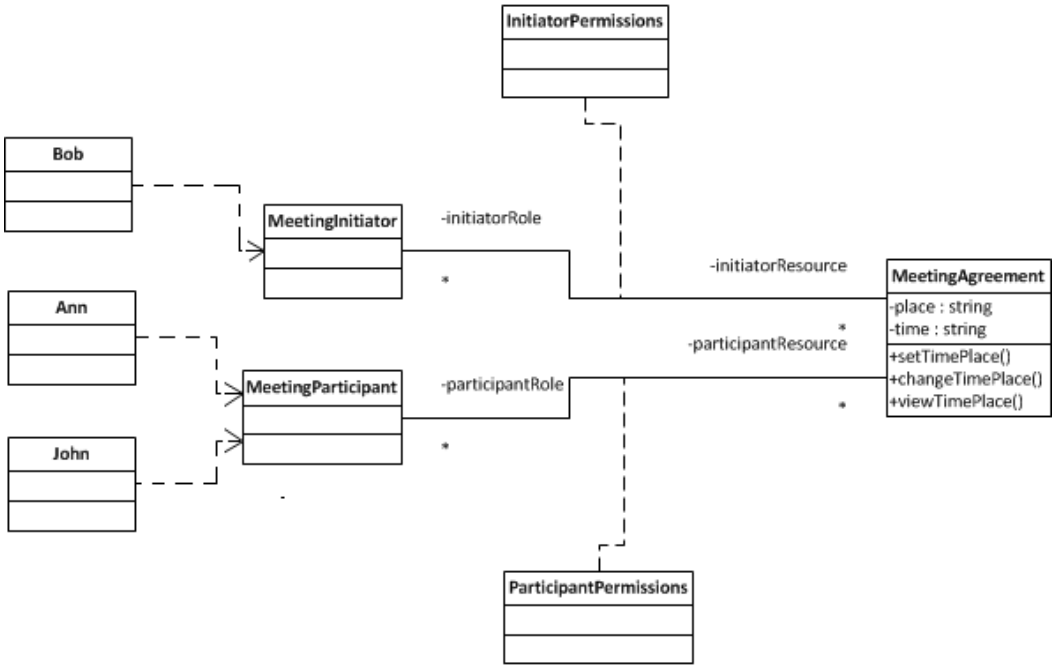

Figure 18 KAOS to SecureUML Transformation Rule # 4

Note1: In KAOS model, there is no authorization constraint therefore, when we generate SecureUML model from KAOS model, after determining the SecureUML permission classes, the developer should write the authorization constraints according to allowed actions in these classes.

Note2: In SecureUML model, there are not only operations (e.g setTimePlace) but also allowed actions in permission classes (e.g changeMeetingInfo: Update). It is easy to take operations from KAOS model but to generate these allowed actions, the developer should benefit from the already existing operations and also from goals. The developer will understand the relationship between these operations and actions. The authorization constraints might help to identify the relationship between them e.g. (*enterAgreementDetails* and *setTimePlace*), and (*getAgreementInformation* and *viewTimePlace*). After that these authorization constraints can be linked to attributes (actions) of permission classes.

Note3: "The SecureUML model needs to be completed manually with the information, which is not captured from the KAOS model. Specifically, the developer needs to introduce the following information:

- the attributes of the <<secuml.resource>> class that define the state of the secured resource(s). For example, the class *MeetingAgreement* should be complemented with attributes *place:String* and *time:String*
- multiplicities for all the association relationships. For example, multiplicities for associations between *MeetingInitiator* and *MeetingAgreement*, *MeetingParticipant* and *MeetingAgreement* have to be defined;
- names for the association classes. For instance, for classes with the <<secuml.permission>> stereotype have to be specified;
- action types for the identified actions. For example, for action *Insert meeting time and place* action type is Insert, for *Update time and place to be suitable*

action type is Update, and for *Check if time and place are suitable* action type is Select." [7].

6.3. Summary

Table 6 shows the comparison of RBAC modeling using SecureUML and KAOS. In order to compare these two security modeling language, we chose the common RBAC concepts which SecureUML and KAOS have.

Table 6 Comparison of RBAC modeling using SecureUML and KAOS

(adapted from [7])

RBAC concepts	**SecureUML**		**KAOS**	
	Construct	**Example**	**Construct**	**Example**
Users (concept)	Class stereotype <<secuml.user>>	Bob, Ann, and John	Entity "User"	"Bob", "Ann", and "John"
User assignment (relationship)	Dependancy stereotype <<assignment>>	Dependancy between classes such as Bob and MeetingInitiator, and Ann or John and MeetingParticipant	User object	Agent >> User >> Name Meeting Initiator >> Bob Meeting Participant >> Ann, John
Roles (concept)	Class stereotype <<secuml.role>>	MeetingInitiator and MeetingParticipant	Agent	MeetingInitiator and MeetingParticipant
Permission assignment (relationship)	Association class stereotype <<secuml.permission>>	InitiatorPermissions and ParticipantPermissions	Performance links	Meeting Initiator <<performance>> Change time and place
Objects (concept)	Class stereotype <<secuml.resource>>	MeetingAgreement	Entity	Meeting Agreement
Operations (concept)	Class operations	setTimePlace(), changeTimePlace(), and viewTimePlace()	An operation	Set time and place, View time and place, and Change time and place
Permissions (concept)	Authorization constraint	AC#1, AC#2, and AC#3	-	Not defined explicitly

Chapter 7. UMLSec - KAOS Transformations

In this chapter, we use a scenario called Meeting Scheduler Example that we already described in Chapter 6. According to this scenario, we create UMLSec diagram and later on we generate transformation rules from UMLSec to KAOS and vice versa.

UMLsec is an extension of UML which allows an application developer to embed security related functionality into a system design and perform security analysis on a model of the system to verify that it satisfies particular security requirements. Security requirements are expressed as constraints on the behavior of the system and the design of the system may be specified either in a UML specification or annotated in source code.

7.1. Meeting Scheduler Example with UMLSec

We use the same meeting scheduler example as we described in Chapter 6 Section 1.

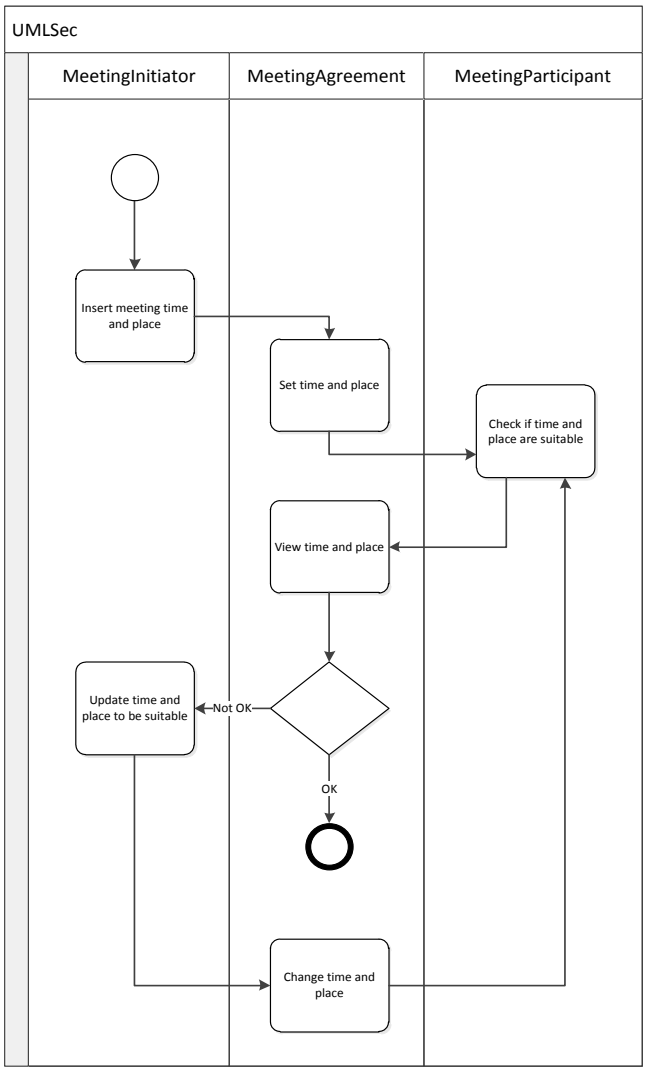

Figure 19 UMLSec diagram for Meeting Scheduler Example (adapted from [7])

There are three associated tags for the protected actions: AT#1, AT#2 and AT#3 to apply security policies.

AT#1 {protected = Set time and date}
{role = (Bob, MeetingInitiator)}
{right = (MeetingInitiator, Set time and place)}

AT#2 {protected = View time and date}
{role = ([Ann, John], MeetingParticipant)}
{right = (MeetingParticipant, View time and place)}

AT#3 {protected = Change time and date}
{role = (Bob, MeetingInitiator)}
{right = (MeetingInitiator, Change time and place)}

7.2. Transformation Rules

There are two sets of transformation rules. First it is from UMLSec to KAOS and the other one is from KAOS to UMLSec.

7.2.1. Model Transformation from UMLSec to KAOS

We will use Fig. 19 UMLSec diagram for meeting scheduler example as our *input*. Below we define four transformation rules to transform a model from UMLSec to KAOS, these are our actions and the final figure that we have Fig. 23 UMLSec to KAOS Transformation Rule # 4 will be our output.

UK1. In the UMLsec model the activity partitions that do not hold secured protected actions, can be transformed to the agents in KAOS model.
Example: We have agents instead of activity lanes in KAOS model. MeetingParticipant and MeetingInitiator.

Figure 20 UMLSec to KAOS Transformation Rule # 1

UK2. Association tags {protected} allow us identify the operations that belong to secured resource. We transform the activity partitions, which hold these operations to the performance relation between operations and the agents who has right to perform these operations in KAOS model.
Example: set time and place, change time and place, and view time and place operations in KAOS model. These operations should be taken from class related activity lanes (see MeetingAgreement). The other actions like insert meeting time and place is not taken place in KAOS model.

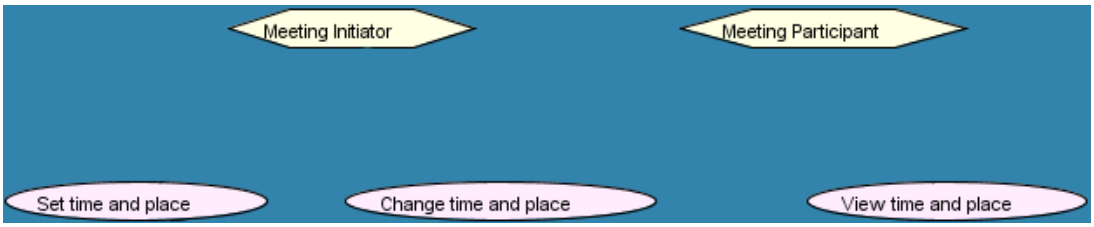

Figure 21 UMLSec to KAOS Transformation Rule # 2

UK3. From UMLSec association tag {right} we are able to identify on which operations a role can perform security actions. Thus, from each occurrence of this association tag in the KAOS model, performance links between agents and corresponding operations are introduced.
Example: UMLSec association tag *right* is handled by performance links in KAOS model. It helps us to understand which operation can be performed by whom. MeetingInitiator have permission to change time and place but MeetingParticipant do not.
{right=(MeetingInitiator, Set time and date)}
{right=MeetingParticipant, View time and date)}
{right=(MeetingInitiator, Change time and date)}

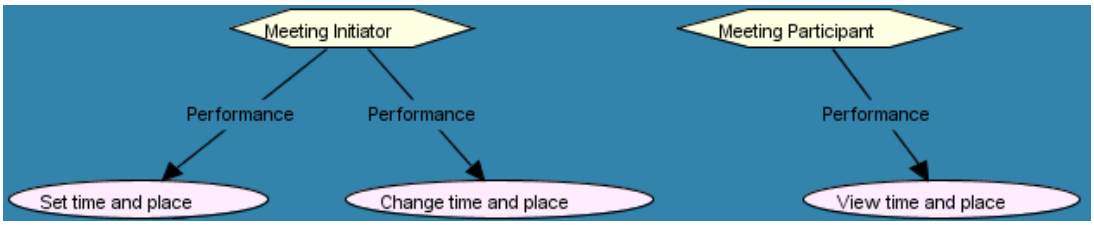

Figure 22 UMLSec to KAOS Transformation Rule # 3

UK4. Association tag {roles} allows us to identify the <<assignment>> dependency relationship between classes of users, in KAOS model they are defined with environment agents which derived from agent entity and their roles presented with agent.
Example: The actor values of associated tag {role} become environment agents. We are assigning environment agents who are responsible from expectations. In my KAOS model, I call them User. It is derived from Agent object. Bob, Ann and John are users.
{role=(Bob, MeetingInitiator)}
{role=([Ann, John], MeetingParticipant)}

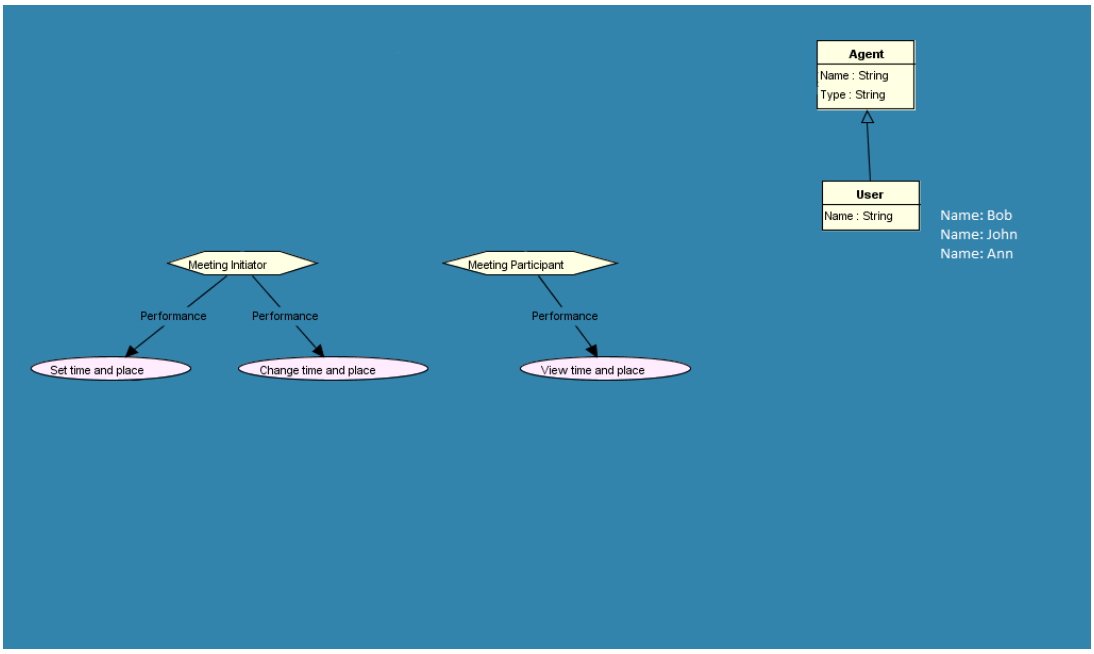

Figure 23 UMLSec to KAOS Transformation Rule # 4

Note1: From UMLSec model, we cannot directly generate the goals that we have to elicit. So, we have to focus the activity partitions and actions to understand the aim and then we may write the goals with some lost information.

Note2: The entities in KAOS object model are derived from activity lanes in UMLSec model. The problem here is some of the activity lanes will replace as agents and some of them will replace as entities in KAOS model. The developer should understand which one is suitable to be agent and which one is suitable to be object. For instance, MeetingInitiator and MeetingParticipant are chosen to be agent and MeetingAgreeement is chosen to be entity.

Note3: The attributes of entities should be filled by the developer. The nouns in the operations will help him to do it. For instance, set time and place action gives him clue that there are two terms whose values may change. These are time and place therefore they became attributes in Meeting Agreement entity. Also, we should link the operations to the entities whose attribute's values depend on the results of these operations. Here we use input/output links.

Note4: In KAOS model, there are association link types, such as *concern, input, output* etc. The developer should analyze the relationship between the constructs and decide to link one to another. The entire list of link types of KAOS model is shown in Table 5.

7.2.2. Model Transformation from KAOS to UMLSec

We will use Fig. 10 meeting scheduler example with KAOS as our *input*. Below we define five transformation rules to transform a model from KAOS to UMLSec, these are our actions and the final figure that we have Fig. 26 KAOS to UMLSec Transformation Rule # 3 will be our output.

KU1. In KAOS model, entities (independent, passive objects) are represented by an activity partition in the UMLsec model.
Example: The agents and objects become activity lanes in UMLSec model. Meeting Agreement entity becomes a lane in UMLSec model.

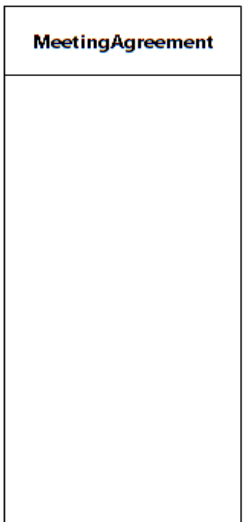

Figure 24 KAOS to UMLSec Transformation Rule # 1

KU2. In KAOS model, agents (independent, active objects) can be transformed to the activity partition in UMLSec model.
Example: Agents are also become activity lanes. MeetingInitiator and MeetingParticipant agents become MeetingInitiator and MeetingParticipant lanes in UMLSec model.

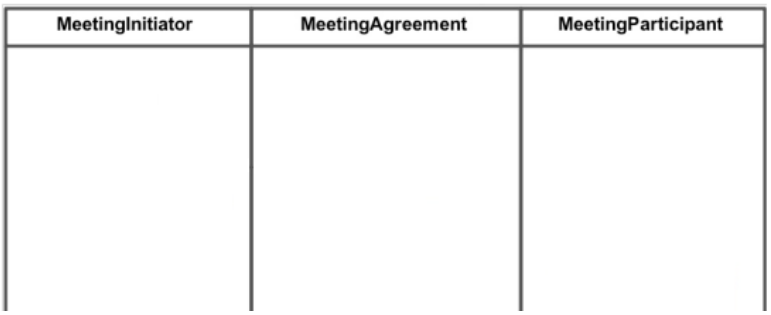

Figure 25 KAOS to UMLSec Transformation Rule # 2

KU3. The operations defined in KAOS model are transformed to actions belonging to this activity partition in UMLSec. In addition, each operation becomes a value the UMLsec associated tag {protected}.
Example: The actions; set time and place, view time and place, change time and place in MeetingAgreement activity lane in UMLSec model.
{protected = Set time and date}
{protected = View time and date}
{protected = Change time and date}

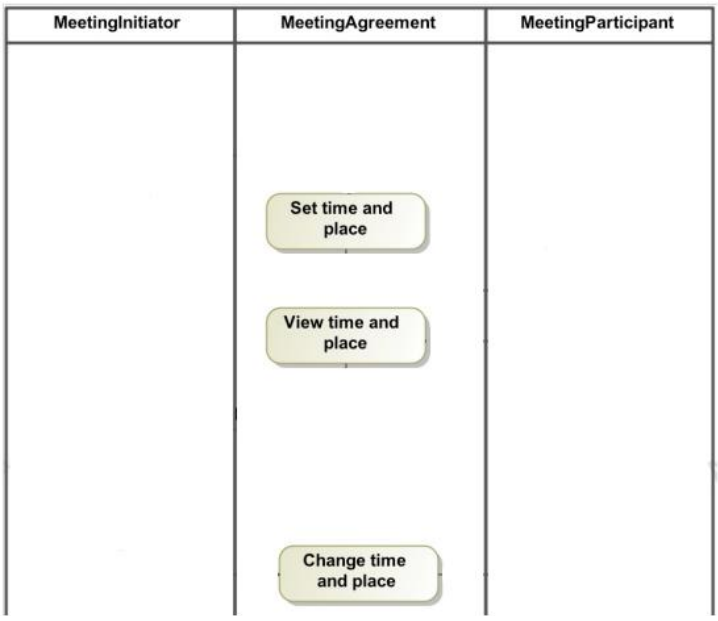

Figure 26 KAOS to UMLSec Transformation Rule # 3

KU4. In KAOS model, users (e.g Bob, Ann and John) are environment agents; they are derived from agent entity. They can be defined with actor value of the associated tag {role} in UMLSec model.
Example: Environment agents become the *Actor* value of the associated tag {role}.
{role=(Bob, MeetingInitiator)}
{role=([Ann, John], MeetingParticipant)}

KU5. In KAOS model, from performance links between agents and operations, we are able to identify on which operations a role can perform security actions. Thus, from each occurrence of this links in the KAOS model, we define the role value for the UMLsec associated tag {right}. The value of right can be formulized in:
{right = (roleName, actionName)}.
Example:
{right=(MeetingInitiator, Set time and date)}
{right=MeetingParticipant, View time and date)}
{right=(MeetingInitiator, Change time and date)}

Note1: "To complete the UMLsec activity diagram a developer needs to specify information that was not possible to capture from the KAOS diagram. For instance the developer needs to define initial node (e.g., to *enterAgreementDetails* action) and activity final node (e.g., from *viewTimePlace* action). Other control flows (including the conditionals ones) need also to be specified. For instance control flows between *setTimePlace* and *getAgreementInformation*, *viewTimePlace* and *changeMeetingInfo*, and *changeTimePlace* and *getAgreementInformation* might define a logical sequence of activity that corresponds to the one in Figure 19." [7].

7.3. Summary

Table 7 shows the comparison of RBAC modeling using UMLSec and KAOS. In order to compare these two security modeling language, we chose the common RBAC concepts which UMLSec and KAOS have.

Table 7 Comparison of RBAC modeling using UMLSec and KAOS

(adapted from [7])

RBAC concepts	UMLSec		KAOS	
	Construct	**Example**	**Construct**	**Example**
Users (concept)	*Actor* value of the associated tag {role}	"Bob", "Ann", and "John"	Entity "User"	"Bob", "Ann", and "John"
User assignment (relationship)	Associated tag {role}	{role=(Bob, MeetingInitiator)} {role=([Ann, John], MeetingParticipant)}	User object	Agent >> User >> Name Meeting Initiator >> Bob Meeting Participant >> Ann, John
Roles (concept)	Role value of the associated tag {role}	"MeetingInitiator" and "MeetingParticipant"	Agent	MeetingInitiator and MeetingParticipant
Permission assignment (relationship)	Associated tag {right}	{right=(MeetingInitiator, Set time and date)} {right=MeetingParticipant, View time and date)} {right=(MeetingInitiator, Change time and date)}	Performance links	Meeting Initiator <<performance>> Change time and place
Objects (concept)	Activity partition	MeetingAgreement	Entity	Meeting Agreement
Operations (concept)	An action	Set time and date, View time and date, and Change time and date	An operation	Set time and place, View time and place, and Change time and place
Permissions (concept)	{role}, {protected}, and {right}	Not defined explicitely	-	Not defined explicitly

PART III

VALIDATION

In Validation Part, we are going to present the type and design of the validation. We will discuss the threats to the validity. According to our validation test, we will validate our results through another scenario called Food Delivery Example. We will compare the transformed models with already existing models.

Chapter 8. Design of Validation and Test

In this chapter, we present our validation type and according to this decision, our test type and its design. Lastly, we will show the threats to the validity.

8.1. Validation Type

"In order for a quality model to be valid, all its metrics (including aggregated metrics and indicators) have to be valid" [8]. Kitchenham *et al*. [3] define two major methods to check metrics validity. (Fig. 27):

- *Theoretical validation*, which confirms that the measurement does not violate any required properties of measurement elements or of the definition models [3].
- *Empirical validation*, which corroborates that measured attributes are consistent with the values predicted by the models involving the attribute [3].

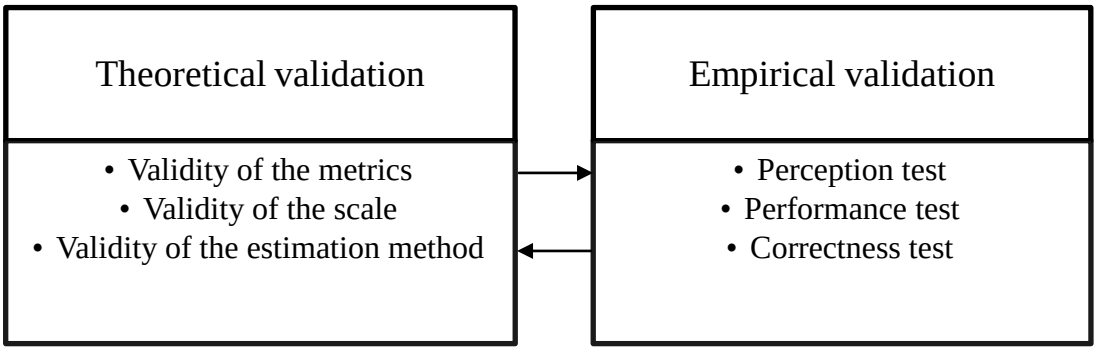

Figure 27 Theoretical and Empirical Validation

(adapted from [3])

There are three different test types as it is shown in Fig. 27. In this work, we focus only empirical validation and its subdivision, correctness test due to the lack of theoretical data. Correctness test is more suitable than perception test and performance test according to check the validity of transformation rules and models because (i) for perception test, we needed a group of people who should use our methodology and validate the work, this option was not convenient to choose (lack of people who is interested in this topic, (ii) for performance test, our main scope is whether we can align KAOS to RBAC or not and how we will do that so this option did not also meet with our expectations. Therefore we apply correctness test to some studies of a specific case.

8.2. Correctness Test

In this section, we present one of empirical validation method, correctness test for KAOS models and transformation rules. This test is to check whether our model and rules are correct or not. In subsection 8.2.1, we describe its correctness test design.

8.2.1. Design

We define transformation rules and use these rules to transform one model to another one. After this transformation, we need to verify the correctness of this transformation. We are doing this in order to validate our work.

The research method: We plan to analyze some case studies focusing on their correctness on specific criteria. Our criteria depend on the transformation rules that we covered in Chapter 6 and Chapter 7. We hope to show that the correctness of the KAOS models and transformation rules are indeed correct and correspond to the models (SecureUML and UMLSec) that we created based on these transformation rules and feedback from the experts mainly my supervisor.

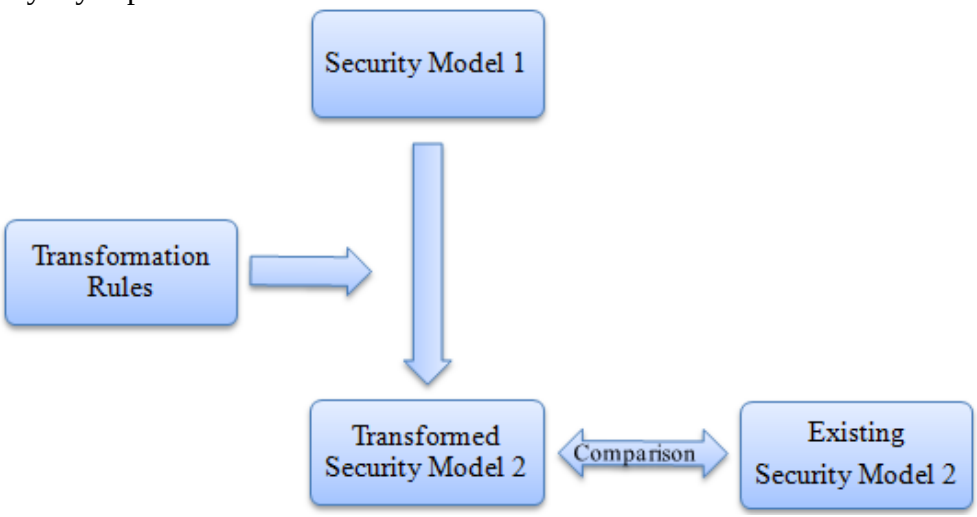

Figure 28 Design of Test

The research sample: We will provide one scenario and create KAOS model based on this case study. After that, we will use the transformation rules to transform this KAOS model to SecureUML model and UMLSec model.

The Scenario:

- Food Delivery Example

8.2.2. Threats to Validity

Before presenting the correctness test results, we discuss some validity threats:

- Reliability of our KAOS model (see Fig. 29) could be seen as the internal validity threat. However, our generated model is theoretically valid. There might be different designs which can lead developer to receive different results.
- Reliability of existing security models (see Fig. 30 and Fig. 31) could be seen as the external validity threat. In order to ensure the accuracy and correctness of these models, I used the related article [7] as a guideline to create these security models.

8.3. Summary

In this chapter, we try to show how to validate and test our approach. Actually we followed an algorithm to achieve this. Here is step by step what to do in order to validate, test and conclude our research.

I. Read scenario or generate your own scenario.
II. Create your security model.
III. Apply transformation rules to this model.
IV. Use other security models manually.
V. Compare transformed and manually created security models.
VI. Report the results.
VII. Update your transformation rules.

Chapter 9. Food Delivery Example

In this chapter, we are going to discuss another example, called food delivery. First of all, we describe the scenario and based on this scenario, we create three models, KAOS, SecureUML and UMLSec respectively. After that, we will apply transformation rules to the models one by one in order get another model. Finally, we will compare the transformed models with the existing models.

9.1. Food Delivery Scenario

The Food delivery example is described as follows: Customer wants to order food to a specific place. He needs to inform the restaurant at appropriate time. He contacts call center agent and sends his request. Call center agent receives the request and transfers it to the courier. The courier delivers the food to the registered address. The Food delivery system helps both customer and call center agent. The customer can use this system to enter his information (name, address, phone, etc.) and see the food information (menus, prices, promotions, available hours to delivery, etc.). The call center agent uses this system to track the orders, online support, and sending notifications.

9.2. KAOS Model

In Figure 29 we present a KAOS model to illustrate RBAC policy for the Food Delivery Example. Here first, we define goals and sub-goals. After that, regarding these goals we defined three agents Courier, CallCenterAgent, and Customer in order to associate each goal with an agent responsible for it. We also present entities which characterize our object model. The notation used in the object model complies with the one used in UML for class diagrams. These entities are FoodOrder and Agent. FoodOrder entity has order related attributes such as orderID, orderStatus, orderAddress, etc. which need to be secured. We use generalization in order to create two objects Software agent and Environment agent. We define three Environment agents Jack, Jane and Mary, who act as a user in the system. Lastly, we define operations Request Order, Cancel Order, Receive Order, and Deliver Order. These operations sum up all the behaviors that agents need to have to fulfill their requirements. Behaviors are expressed in terms of operations performed by agents. With KAOS, the operations are connected to the goals, we justify operations by the goals they "operationalize".

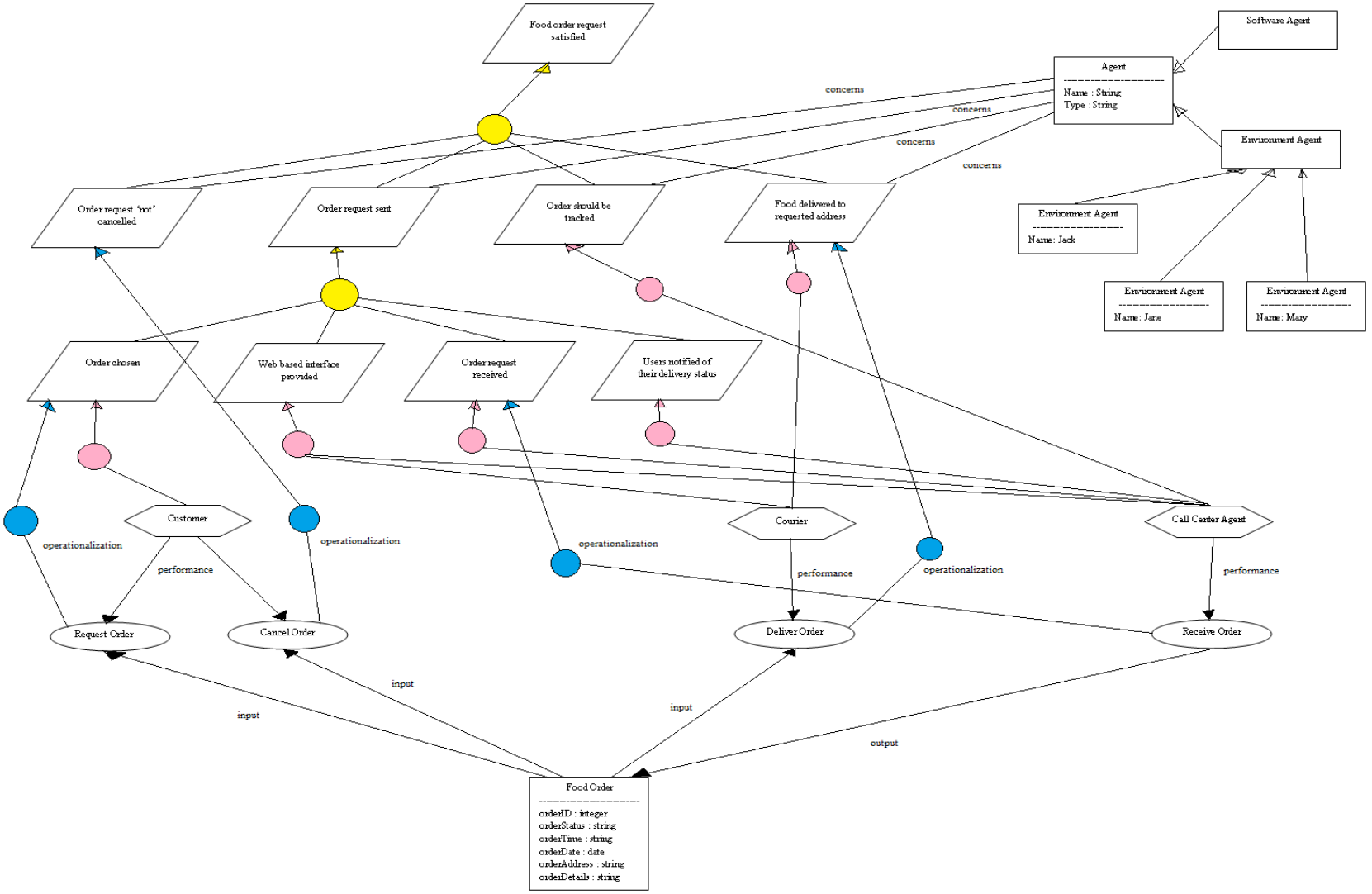

Figure 29 Food Delivery Example KAOS Model

9.3. SecureUML Model

In Figure 30 we present a SecureUML model to illustrate RBAC policy for the Food Delivery Example. Here we define three users Jack, Jane and Mary, who play different roles in the system. We also present that a resource (FoodOrder), which characterize order attributes (orderID, orderStatus, orderTime, etc.) of the food delivery, needs to be secured. Thus, a certain restriction on changing the value of the attributes of this resource needs to be defined for the role Courier, CallCenterAgent and Customer.

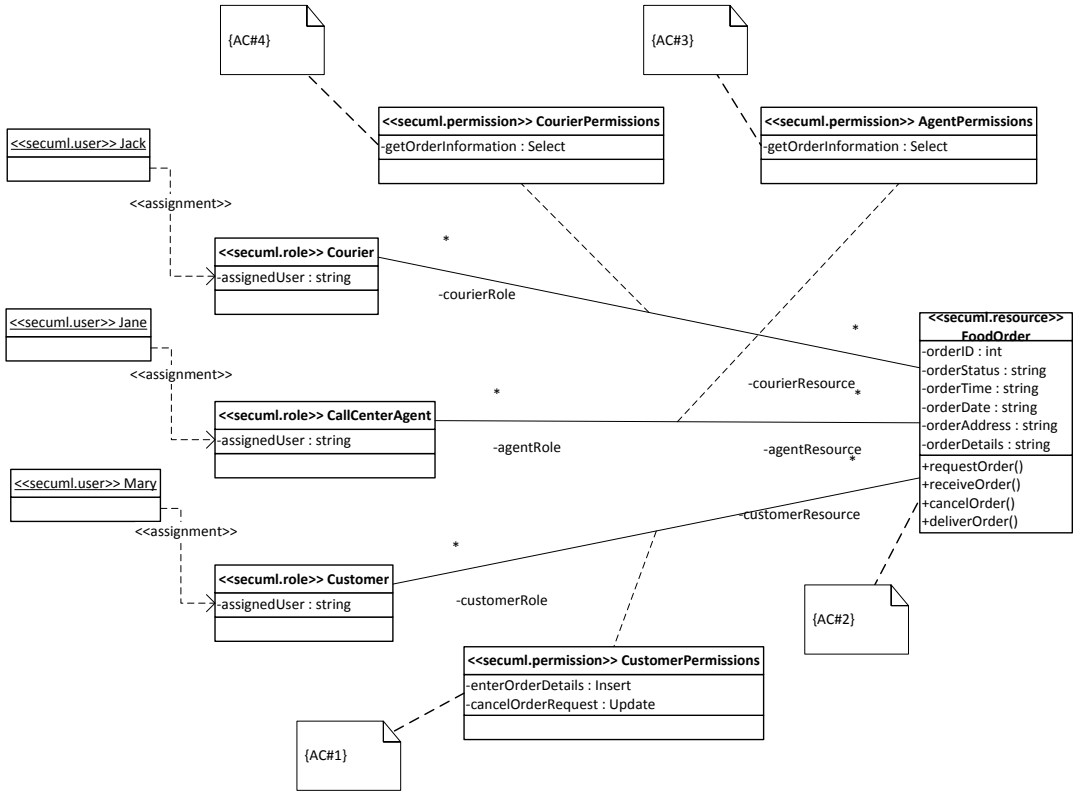

Figure 30 Food Delivery Example SecureUML Model

Association class CustomerPermissions characterizes two actions allowed for the Customer. Action enterOrderDetails (of type Insert) defines that Customer can enter order attributes by executing operation requestOrder() (see class FoodOrder), and action cancelOrderRequest (of type Update) allows changing status of the FoodOrder by executing operation cancelOrder() (see class FoodOrder). To strengthen these permissions we define authorization constraints AC#1 and AC#2. Authorization constraint AC#1 means that operation requestOrder() (of class FoodOrder) can be executed by one user Mary assigned to a role Customer. Likewise, the authorization constraint AC#2 defines restriction for operation cancelOrder() (of class FoodOrder):

AC#1 context FoodOrder::requestOrder():void
pre: self.roleCustomer.assignedUser ->
exists(i | i.assignedUser = “Mary”)

```
AC#2    context FoodOrder::cancelOrder():void
        pre: self.roleCustomer.assignedUser ->
                exists(i | i.assignedUser = "Mary")
```

Association class AgentPermissions defines a restriction for the CallCenterAgent role. It defines an action getOrderInformation (of type Select) that says that only CallCenterAgent can receive (view) order information defined in the FoodOrder. To enforce this permission an authorization constraint AC#3 is defined:

Authorization constraint AC#3 says that only user Jane who has an assigned role CallCenterAgent can execute an operation receiveOrder() (of class FoodOrder).

```
AC#3    context FoodOrder::receiveOrder():void
        pre: self.roleAgent.assignedUser ->
                exists(i | i.assignedUser = "Jane")
```

Association class CourierPermissions defines a restriction for the Courier role. It defines an action getOrderInformation (of type Select) that says that only Courier can deliver order. In order to do this, he needs order information defined in the FoodOrder. To enforce this permission an authorization constraint AC#4 is defined:

Authorization constraint AC#4 says that only user Jack who has an assigned role Courier can execute an operation deliverOrder() (of class FoodOrder).

```
AC#4    context FoodOrder::deliverOrder():void
        pre: self.roleCourier.assignedUser ->
                exists(i | i.assignedUser = "Jack")
```

9.4. UMLSec Model

Figure 31 illustrates application of UMLSec to model the Food Delivery Example. Here we define an activity diagram, which describes an interaction between Customer, FoodOrder, CallCenterAgent, and Courier. The diagram specifies that Customer can insert order details. Next CallCenterAgent is able to check if the order details are valid and suitable. If Customer wants to cancel his order, he can do it. Otherwise Courier checks the order details in order to finalize the delivery.

This diagram carries an <<rbac>> stereotype, meaning that the security policy needs to be applied to the protected actions. For instance, the Customer's action Insert order details leads to the action Request order for the FoodOrder. Request order is executed if and only if there exists an associated tag, that defines the following: (i) Request order is a protected action, (ii) Mary plays a role of Customer, and (iii) Customer enforces the action Request order. In the activity diagram this associated tag (AT#1) is defined as follows:

```
AT#1    {protected = Request order}
        {role = (Mary, Customer)}
        {right = (Customer, Request order)}}
```

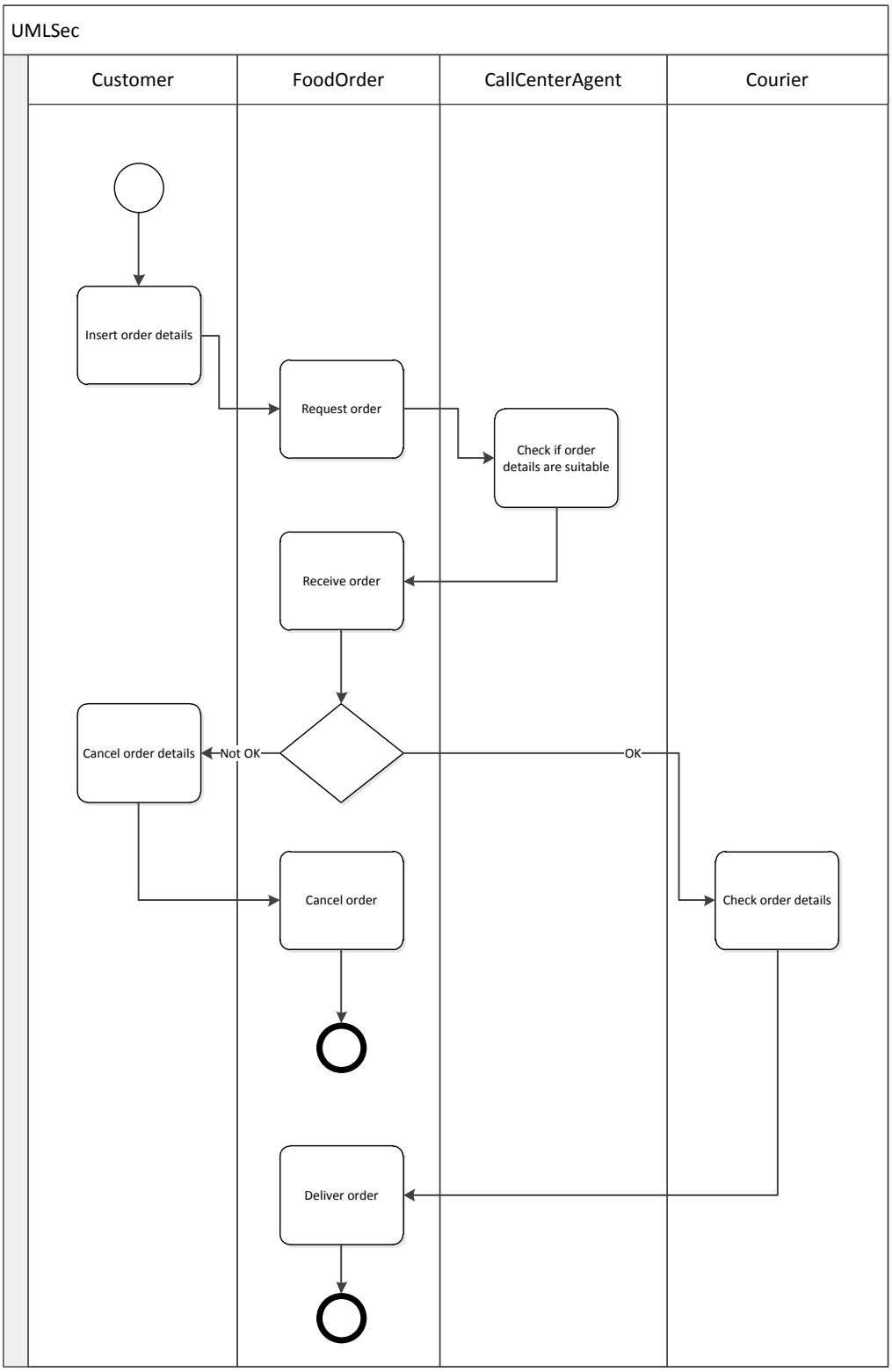

Figure 31 Food Delivery Example UMLSec Model

Similarly, the sets of associated tags are defined for other three protected actions Receive order (AT#2), Cancel order (AT#3), and Deliver order (AT#4).

AT#2 {protected = Receive order}
{role = (Jane, CallCenterAgent)}
{right = (CallCenterAgent, Receive order)}}

AT#3 {protected = Cancel order}
{role = (Mary, Customer)}
{right = (Customer, Cancel order)}}

AT#4 {protected = Deliver order}
{role = (Jack, Courier)}
{right = (Courier, Deliver order)}}

9.5. Applying Transformation Rules

In this section we are going to apply transformation rules to the models that we have created in KAOS, SecureUML and UMLSec. We will get the following outputs according to our inputs:

- KAOS → $SecureUML_1$
- KAOS → $UMLSec_1$
- SecureUML → $KAOS_1$
- UMLSec → $KAOS_2$

9.5.1. KAOS to SecureUML$_1$

We will use as our input Fig. 29 food delivery example KAOS model in order to get a SecureUML model using transformation rules that we already covered in Chapter 6.

KS1. FoodOrder entity became FoodOrder resource class in SecureUML.

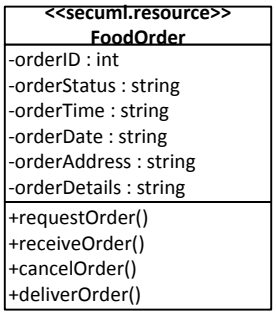

Figure 32 KAOS to SecureUML Transformation Step 1

KS2. Agents; Courier, Call Center Agent and Customer became roles with the same names in SecureUML. These roles are linked to the FoodOrder resource class with binary associations.

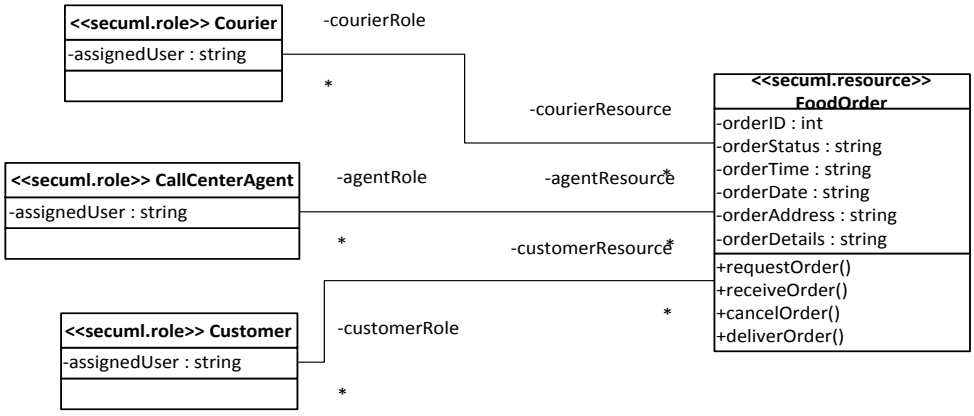

Figure 33 KAOS to SecureUML Transformation Step 2

KS3. Environment agents (users); Jack, Jane and Mary became users with the same names in SecureUML. They are assigned to the roles with assignment links.

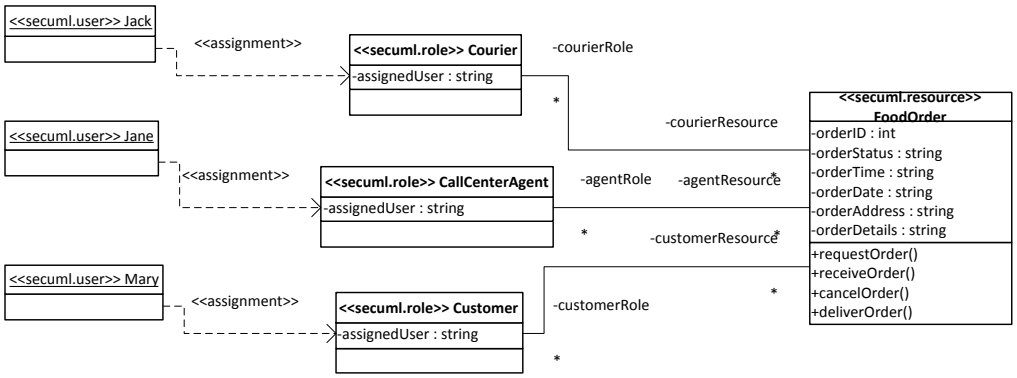

Figure 34 KAOS to SecureUML Transformation Step 3

KS4. Performance links between agents and operations are replaced with permission associated classes.

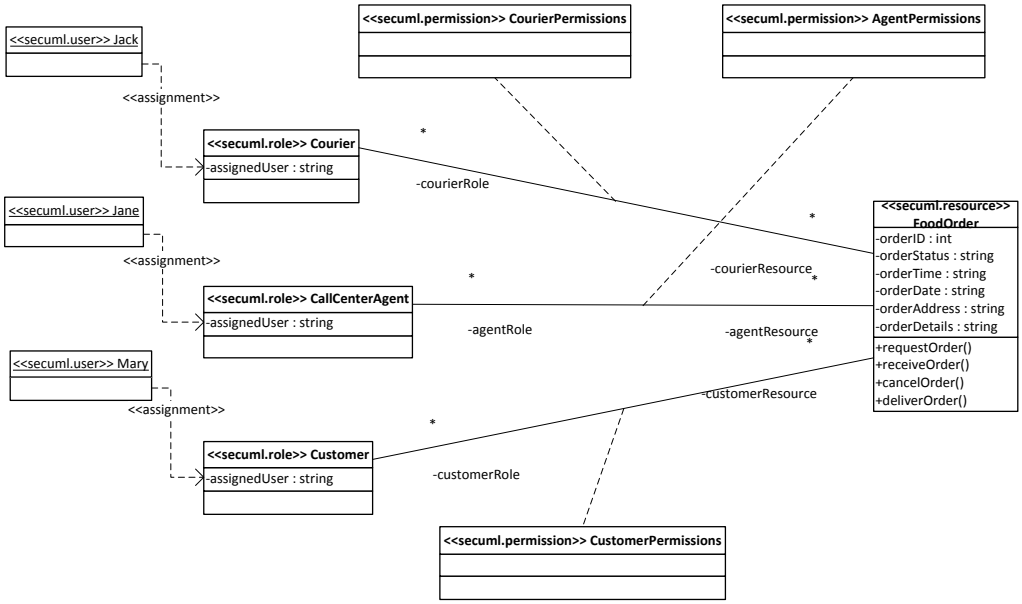

Figure 35 KAOS to SecureUML Transformation Step 4

Note1:

AC#1 context FoodOrder::requestOrder():void
pre: self.roleCustomer.assignedUser ->
exists(i | i.assignedUser = “Mary”)

AC#2 context FoodOrder::cancelOrder():void
pre: self.roleCustomer.assignedUser ->
exists(i | i.assignedUser = “Mary”)

AC#3 context FoodOrder::receiveOrder():void
pre: self.roleAgent.assignedUser ->
exists(i | i.assignedUser = “Jane”)

AC#4 context FoodOrder::deliverOrder():void
pre: self.roleCourier.assignedUser ->
exists(i | i.assignedUser = “Jack”)

Note2:
We have to determine the attributes (actions) of permission classes. In order to do that, we should analyze the authorization constraints and define actions according to these operations. The names of these actions can be synonyms of the operation names or represent the same meaning of those operations.

requestOrder() → enterOrderInformation
cancelOrder() → cancelOrder
receiveOrder() → takeOrder
deliverOrder() → -

Note3:
We need to introduce the following information manually:

- the attributes of the <<secuml.resource>> class that define the state of the secured resource(s).
- multiplicities for all the association relationships.
- names for the association classes.
- action types for the identified actions.

SecureUML$_1$ Model:

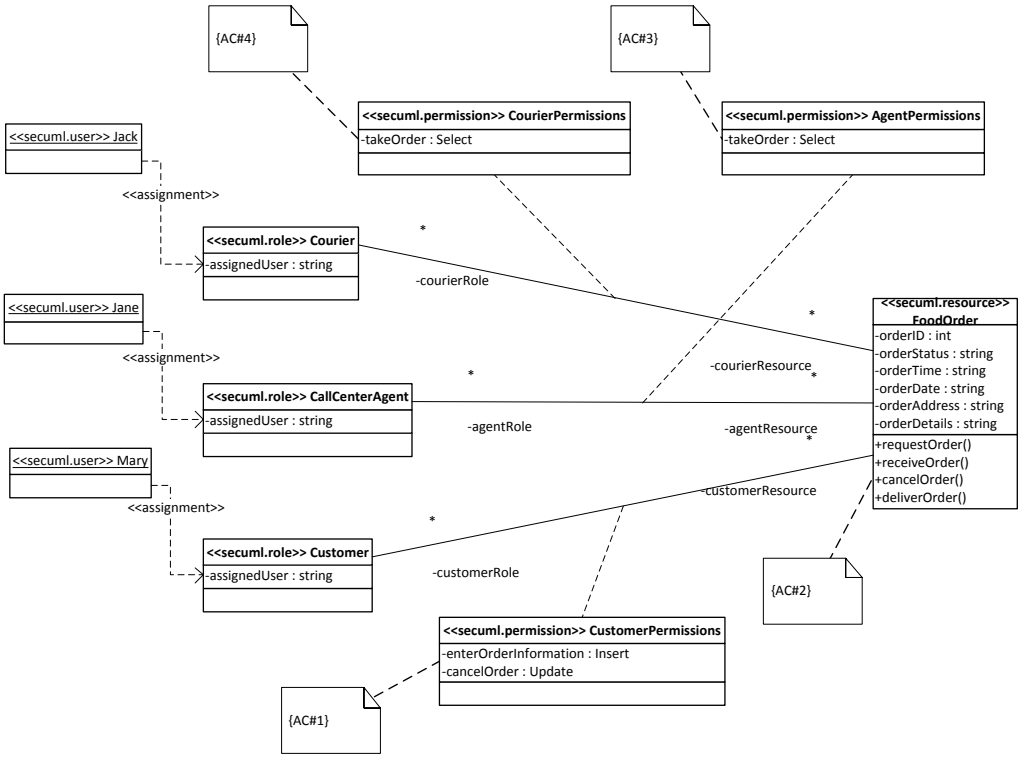

Figure 36 SecureUML$_1$ Model

9.5.2. KAOS to UMLSec$_1$

We will use as our input Fig. 29 food delivery example KAOS model in order to get a UMLSec model using transformation rules that we already covered in Chapter 7.

KU1. FoodOrder entity became activity lane in UMLSec.

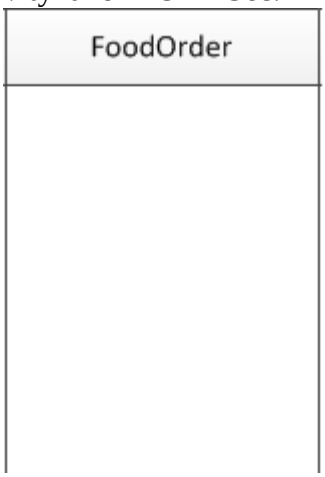

Figure 37 KAOS to UMLSec Transformation Step 1

KU2. Agents; Customer, Call Center Agent and Courier became activity lanes in UMLSec.

Figure 38 KAOS to UMLSec Transformation Step 2

KU3. The operations in KAOS model became protected actions in UMLSec and they are placed in FoodOrder activity lane.
{protected = Request order}
{protected = Receive order}
{protected = Cancel order}
{protected = Deliver order}

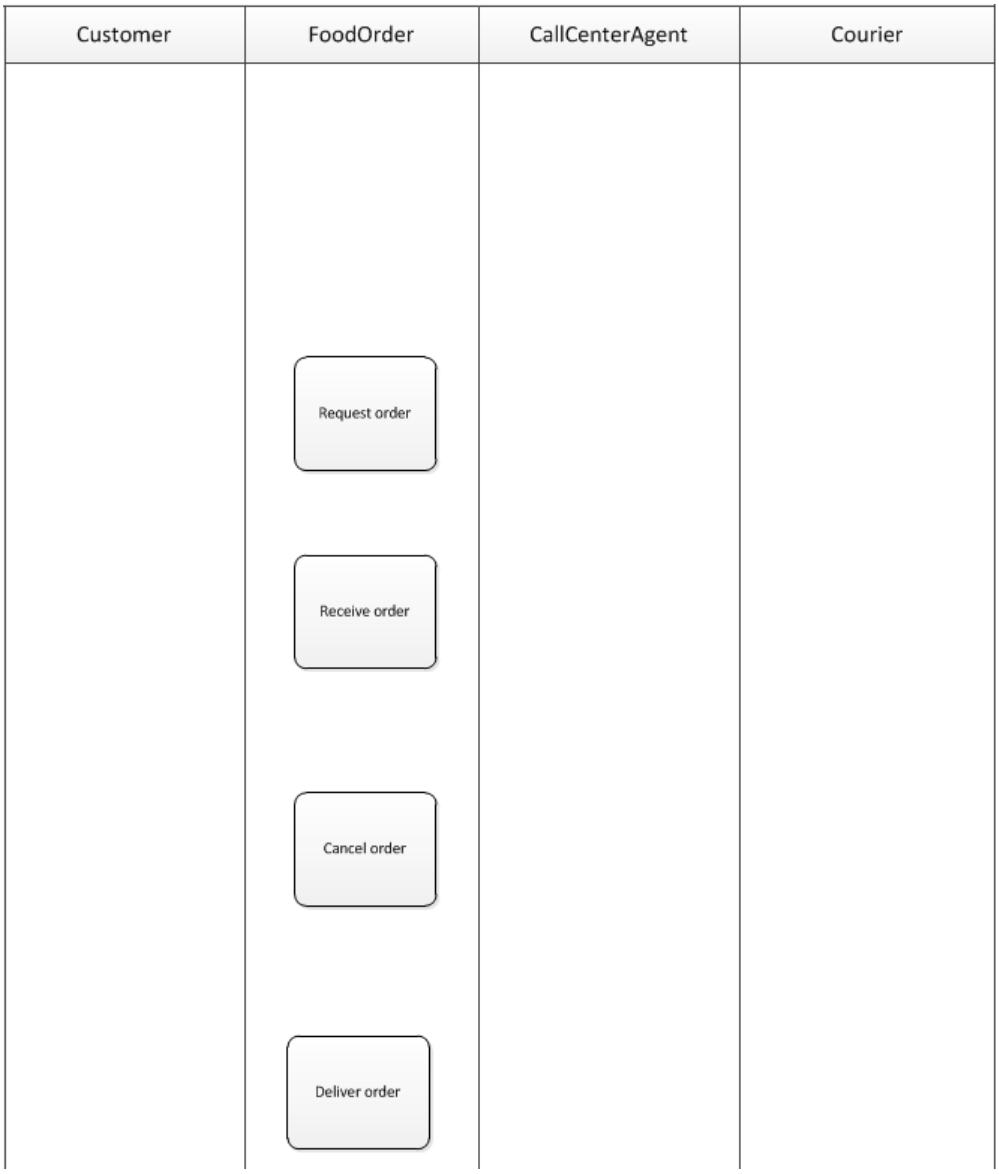

Figure 39 KAOS to UMLSec Transformation Step 3

KU4. The associated tag, {role} is assigned to actorName(s) and roleName(s).
{role = (Mary, Customer)}
{role = (Jane, CallCenterAgent)}
{role = (Mary, Customer)}
{role = (Jack, Courier)}

KU5. Another associated tag, {right} is assigned to roleName(s) and actionName(s).
{right = (Customer, Request order)}}
{right = (CallCenterAgent, Receive order)}}
{right = (Customer, Cancel order)}}
{right = (Courier, Deliver order)}}

Note1:
Since UMLSec is an activity diagram, we have to specify additional information that we cannot capture from KAOS diagram directly. These are initial node, final node, conditional flows and other control flows of activity diagram.

UMLSec$_1$ Model:

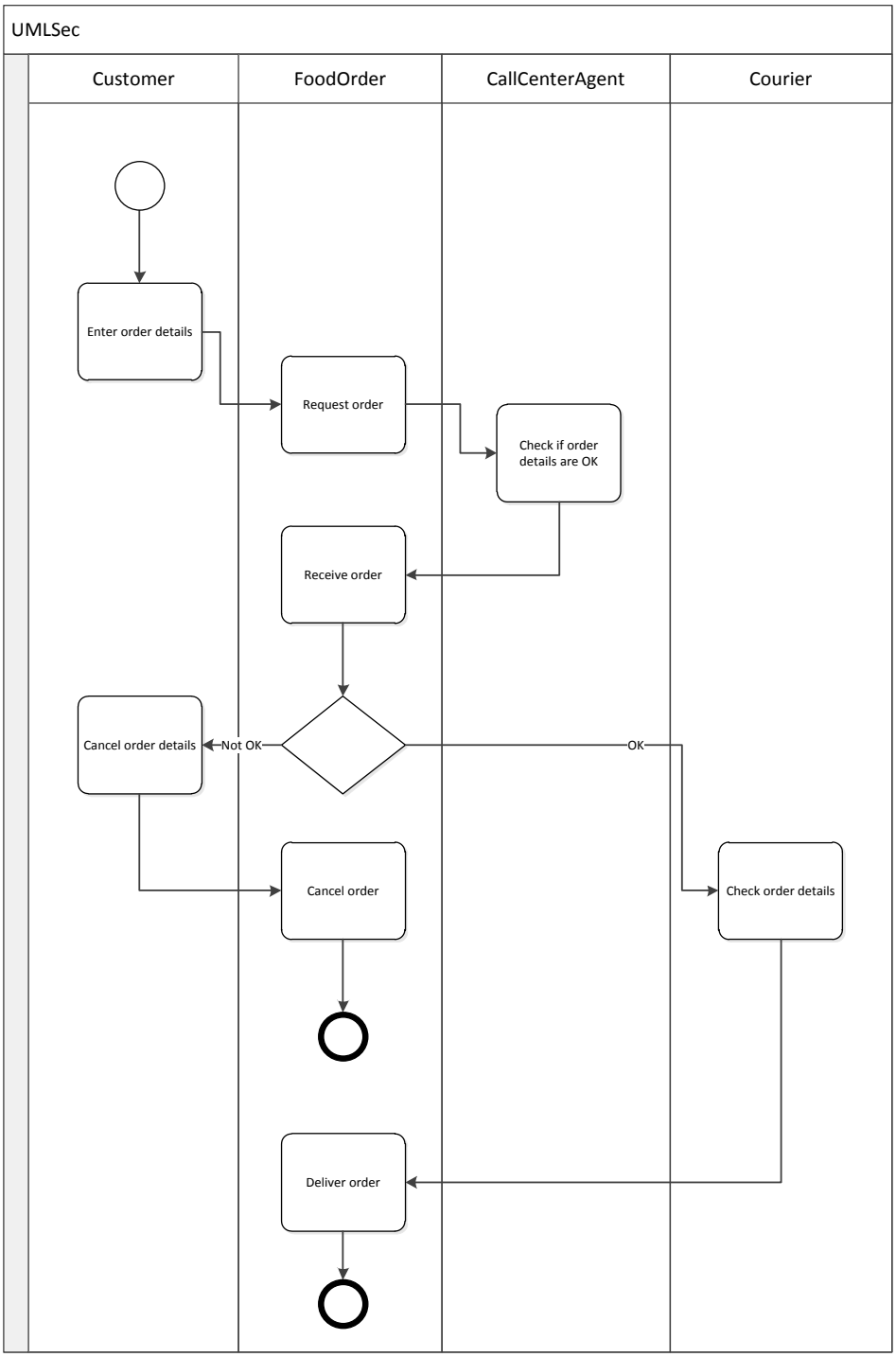

Figure 40 UMLSec$_1$ Model

9.5.3. SecureUML to KAOS$_1$

We will use as our input Fig. 30 food delivery example SecureUML model in order to get a KAOS model using transformation rules that we already covered in Chapter 6.

SK1. SecureUML role classes; Customer, Courier and Call Center Agent became agents in KAOS model.

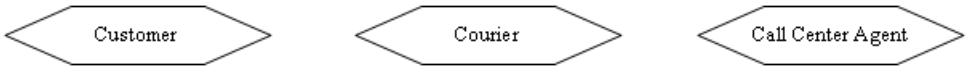

Figure 41 SecureUML to KAOS Transformation Step 1

SK2. SecureUML permission classes became performance links between the agents (Customer, Courier and Call Center Agent) and the operations (Request Order, Cancel Order, Deliver Order and Receive Order) in KAOS model.

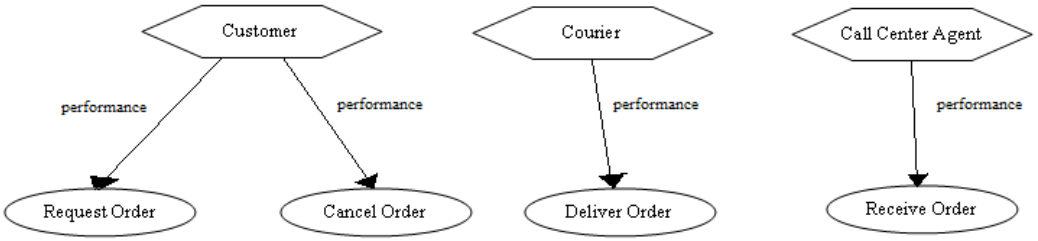

Figure 42 SecureUML to KAOS Transformation Step 2

SK3. SecureUML resource class, FoodOrder became FoodOrder entity in KAOS model.

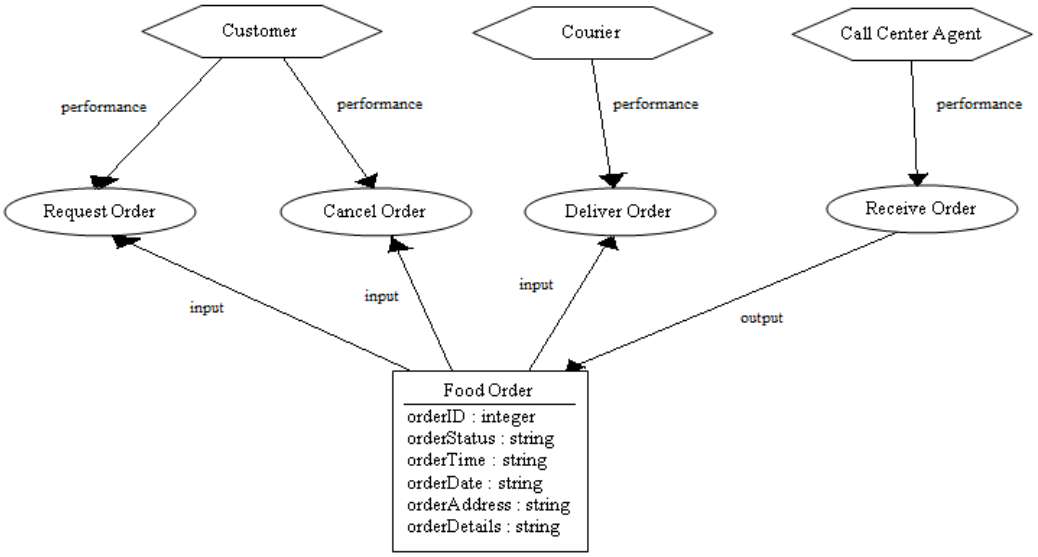

Figure 43 SecureUML to KAOS Transformation Step 3

SK4. SecureUML user classes; Jack, Jane and Mary became Environment agents in KAOS model.

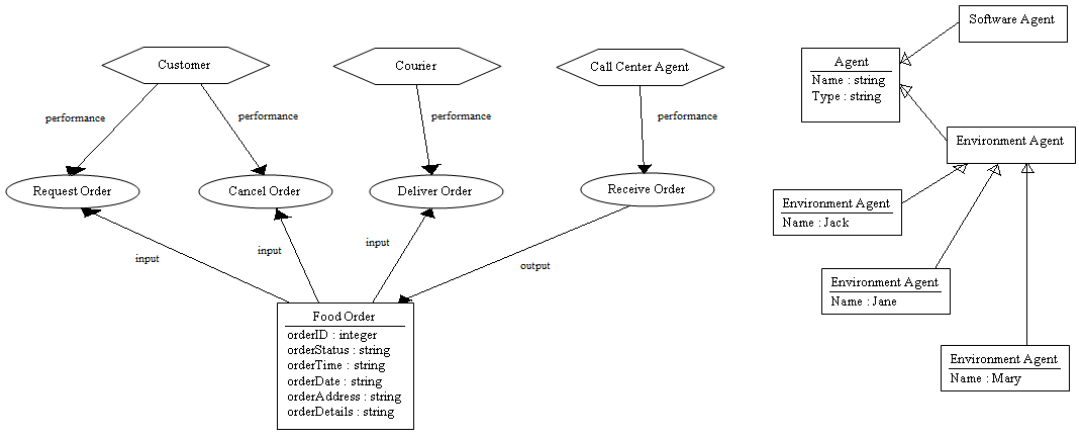

Figure 44 SecureUML to KAOS Transformation Step 4

Note1:
We cannot generate the goals directly from SecureUML model. Therefore, we can discover the goals by interviewing the users, by analyzing the scenario and reading available technical document. This means that goals elicitation cannot be automatically done.

Note2:
We should analyze the relationship between the constructs and decide to link one to another.

$KAOS_1$ Model:

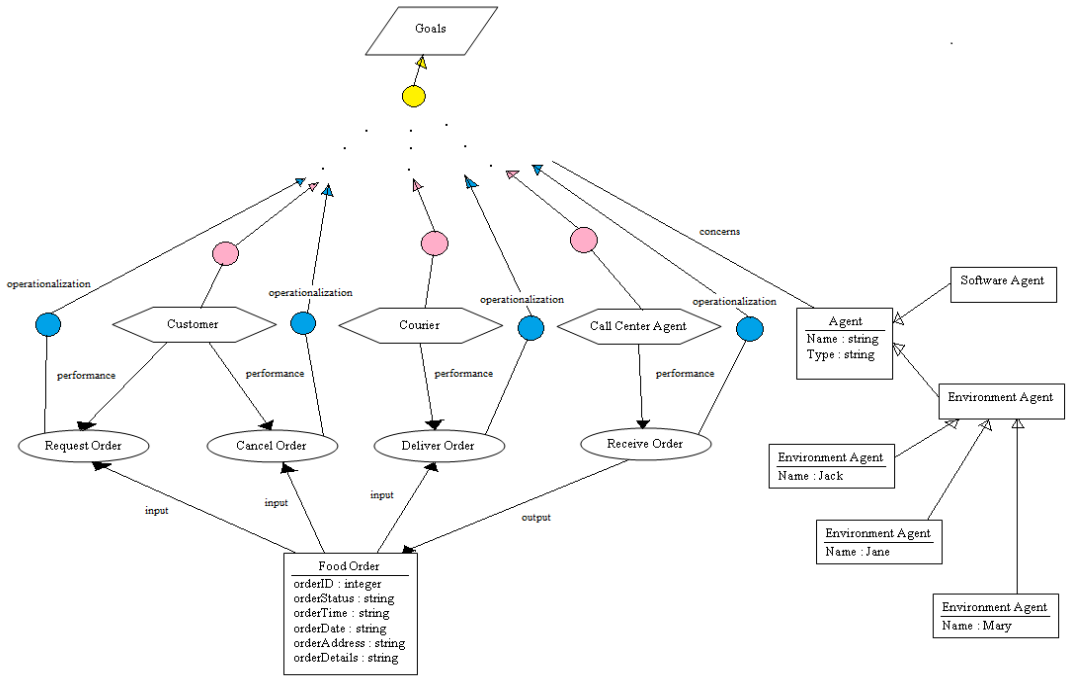

Figure 45 $KAOS_1$ Model

9.5.4. UMLSec to KAOS$_2$

We will use as our input Fig. 31 food delivery example UMLSec model in order to get a KAOS model using transformation rules that we already covered in Chapter 7.

UK1. Activity partitions; Customer, Courier and Call Center Agent became agents in KAOS model.

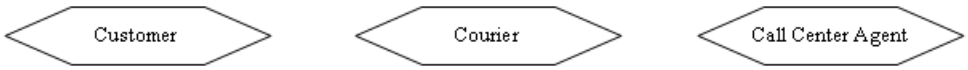

Figure 46 UMLSec to KAOS Transformation Step 1

UK2. Protected actions became operations in KAOS model.

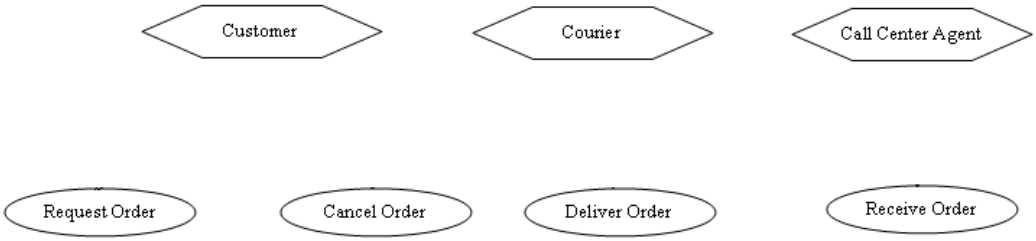

Figure 47 UMLSec to KAOS Transformation Step 2

UK3. The role names and action names which are represented with associated tag {right} became performance links between agents and operations.

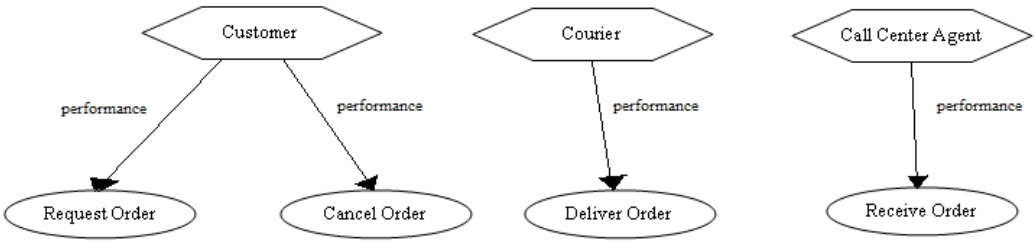

Figure 48 UMLSec to KAOS Transformation Step 3

UK4. The actor names and role names which are represented with associated tag {role} became Environment agents in KAOS model.

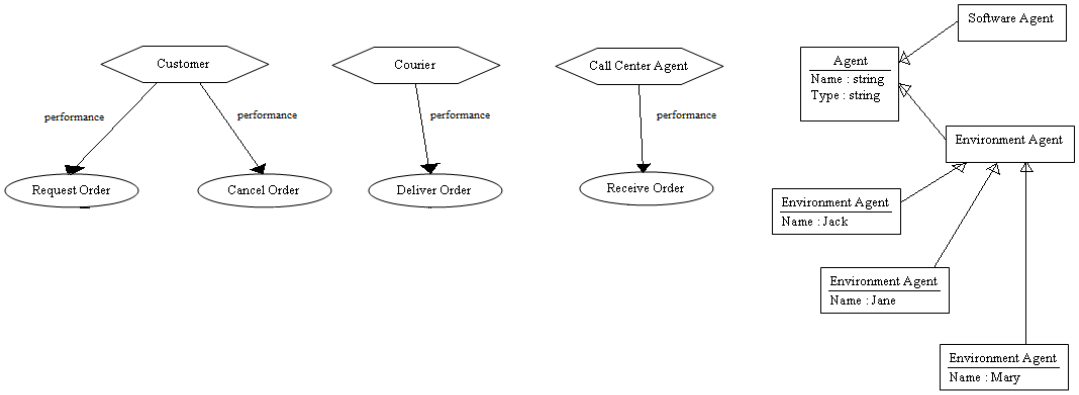

Figure 49 UMLSec to KAOS Transformation Step 4

Note1:
We cannot generate the goals directly from UMLSec model. Therefore, we can discover the goals by interviewing the users, by analyzing the scenario and reading available technical document. This means that goals elicitation cannot be automatically done.

Note2:
We should create entities from activity lanes in UMLSec model. The problem here is some of the activity lanes will replace as agents and some of them will replace as entities in KAOS model. The developer should understand which one is suitable to be agent and which one is suitable to be object. For instance, Food Order represents an entity in KAOS model.

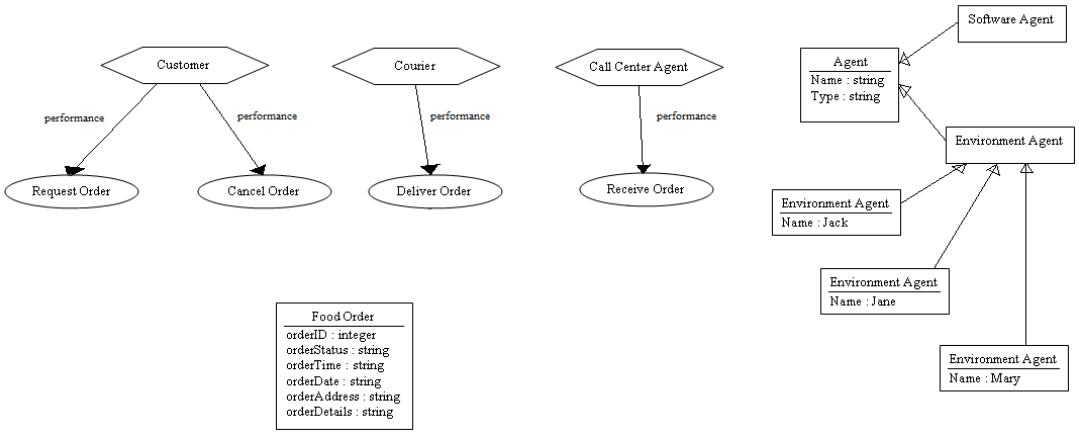

Figure 50 UMLSec to KAOS Transformation Note 2

Note3:
The attributes of entities should be filled by us as well. Also, we should link the operations to the entities whose attribute's values depend on the results of these operations. Here we use input/output links.

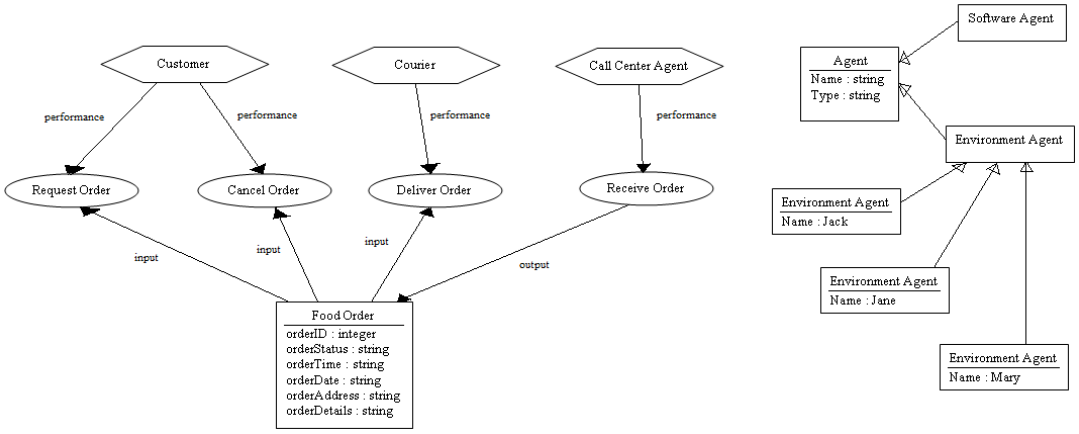

Figure 51 UMLSec to KAOS Transformation Note 3

Note4:
We should analyze the relationship between the constructs and decide to link one to another.

$KAOS_2$ Model:

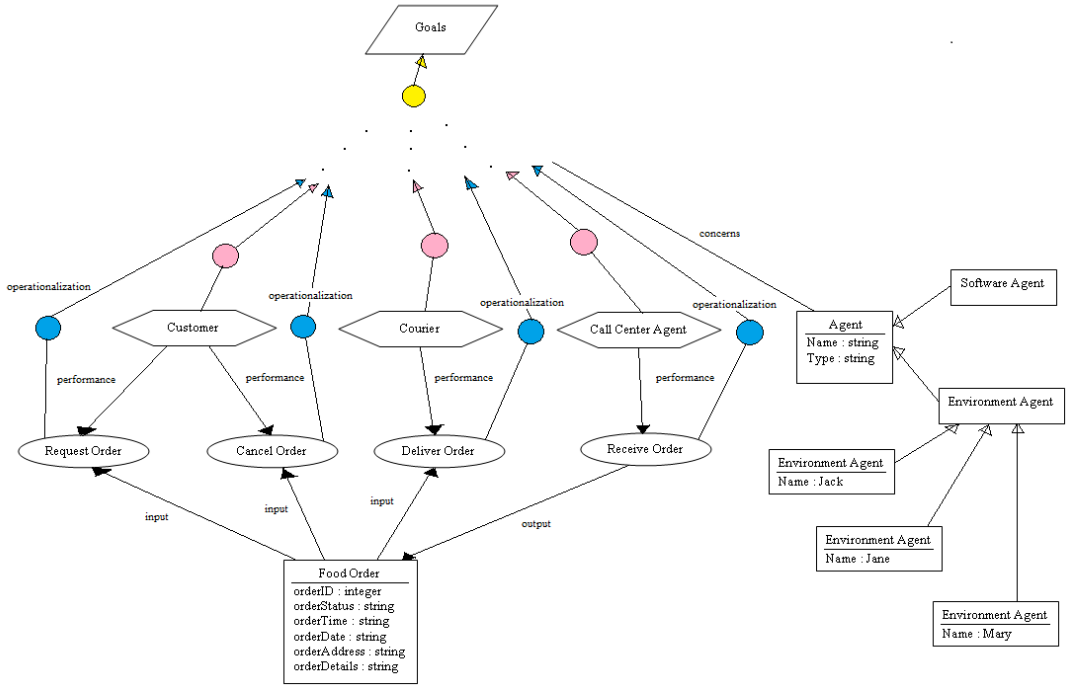

Figure 52 $KAOS_2$ Model

As it is seen on $KAOS_1$ and $KAOS_2$ models. They are exactly same.

9.6. Comparison of Models

In this section, we are going to compare the models according to their correctness. We are going to make the following comparisons:

- KAOS vs. $KAOS_1$
- KAOS vs. $KAOS_2$
- SecureUML vs. $SecureUML_1$
- UMLSec vs. $UMLSec_1$

9.6.1. KAOS vs. KAOS$_1$

Table 8 KAOS vs. KAOS$_1$

	Differences
KAOS - KAOS$_1$	1. Goals - Except the other constructs we could not generate the goals automatically from SecureUML and UMLSec models. 2. Links - The links between some of the constructs are easy to determine but especially the links between the goals and the others could not been generated.

Table 8 shows the differences between KAOS and KAOS$_1$. The goals are desired system properties that have been expressed by some stakeholder(s). So we have to elicit the goals with the help of the stakeholders and especially the users of the system. Also the user scenarios/stories and the other technical documentation might help developer to determine the goals. The links related to the goals such as *operationalization, concern,* etc. can be done easily after goal elicitation just the developer should understand the relationship between the constructs carefully.

9.6.2. KAOS vs. KAOS$_2$

Table 9 KAOS vs. KAOS$_2$

	Differences
KAOS - KAOS$_2$	1. Goals - Except the other constructs we could not generate the goals automatically from SecureUML and UMLSec models. 2. Links - The links between some of the constructs are easy to determine but especially the links between the goals and the others could not been generated.

Table 9 shows the differences between KAOS and KAOS$_2$. The goals are desired system properties that have been expressed by some stakeholder(s). So we have to elicit the goals with the help of the stakeholders and especially the users of the system. Also the user scenarios/stories and the other technical documentation might help developer to determine the goals. The links related to the goals such as *operationalization, concern,* etc. can be done easily after goal elicitation just the developer should understand the relationship between the constructs carefully.

9.6.3. SecureUML vs. SecureUML$_1$

Table 10 SecureUML vs. SecureUML$_1$

	Differences
SecureUML - SecureUML$_1$	1. Actions' names - Actions' names are different. This will not cause a problem just they should correspond to operations logically. 2. Authorization constraints - Linking authorization constraints to actions/operations is complicated. Since I am the only one who designs these models, according to my design I did it in this way.

Table 10 shows the differences between SecureUML and SecureUML$_1$. The developer should define comprehensible and logical names for the actions. Also the developer should give correct action type to these actions (Select, Update, Insert, etc.). The developer should be very careful about this linking authorization constraints to actions/operations. Some of these links can be added to permission classes and some of them can be added to resource classes. This is designer's decision.

9.6.4. UMLSec vs. UMLSec$_1$

Table 11 UMLSec vs. UMLSec$_1$

	Differences
UMLSec - UMLSec$_1$	1. Naming - Actually the transformation was very successful just there are some naming differences. 2. Conditional - Control flows - In comparison there was no difference but general usage of conditional and control flows depend on design. So it might show some difference in activity diagram from design to design.

Table 11 shows the differences between SecureUML and SecureUML$_2$. In order to minimize the mistakes, we have to name the activity names relevant to the scenario and operations. Since UMLSec is an activity diagram, we have to specify some additional information that we cannot capture from KAOS diagram directly. These are initial node, final node, conditional flows and other control flows of activity diagram. It needs to be written manually by the developer.

9.7. Summary

In this chapter, we focus on to a specific example, called food delivery scenario. Based on the information provided to us, we first create our own security models in KAOS, SecureUML and UMLSec. After that, we applied the transformation rules that we already covered in Chapter 6 and Chapter 7 to these security models in order to get semi-automatically transformed ones. Finally, we made comparison regarding to their correctness between transformed models and manually created model.

PART IV

CONCLUSION

In Conclusion Part, we are going to finalize our thesis. After discussing our results, we will talk about some limitations and future work.

After conclusion part, you can find resümee (Estonian translation of the abstract) and references.

Chapter 10. Conclusion

In this chapter, we will conclude our work regarding the research done in this thesis. After showing the results, we will talk about the future work.

In this research we have analyzed how KAOS can help defining security issues through the role-based access control mechanism. The contribution of this study gives modelers the criteria (which modeling approach meets the expectations and satisfies the needs of RBAC) whether they should select KAOS for the RBAC analysis or not.

Our major conclusions include the following:

- We observe that KAOS is applicable to model RBAC solutions. Table 6 and Table 7 both illustrate that KAOS, SecureUML and UMLSec approaches have means to address the RBAC concepts and relationships. Besides, Figure 8 shows that some of the elements in KAOS metamodel such as environment agent, agent and operation, etc. are part of RBAC which refer to user, role and operation respectively.

- According to the results, our transformation rules are not enough to get correct models, they are beneficial but the information system developers and designers should also involve in the transformation phase. These transformation rules helped us to show how we aligned KAOS to RBAC. Here, transformation rules are involved to make the alignment between KAOS and RBAC usable.

This study is not without limitations. Firstly, we should say that our analysis is of limited scope, as it is only based on the literature work [7] and on two simple examples, Meeting Scheduler Example [16] and our own created Food Delivery Example. If we carried out an extensive study or a set of examples we could receive different results. Secondly, the transformation rules that we introduced do not provide automatic transformation because transformation rules rely on to the examples (Meeting Scheduler and Food Delivery) which we used. That's why in another research with different examples, the definition of the transformation rules can be obtained differently.

In this work we did not have a scope to define model transformation rules between KAOS-SecureUML, KAOS-UMLSec and vice versa. However, in our contribution we include a set of guidelines in other words transformation rules (see Chapter 6 and Chapter 7) that could facilitate preparation of the RBAC activity diagrams, if one of these security modeling diagrams (KAOS, SecureUML or UMLSec) is already being defined. But we also should acknowledge that these transformation rules, currently should not be taken for granted because a further and more detailed analysis is required in order to define automatically transformation between the security modeling approaches. Such a definition remains for future work. Since the transformations could not be done automatically, maybe a tool for these languages might be implemented.

RESÜMEE

Turvalisust peetakse infosüsteemide üheks aspektiks. RBAC on lähenemine, mis piirab süsteemi ligipääsu ainult autoriseeritud kasutajatele infosüsteemides. Olemasolevad turvalisusmudelite keeled või lähenemised adresseerivad IS-i turvalisust, kuigi olemasolevad keeled või lähenemised tingimata ei kohandu RBAC-i vajadustele. On olemas mitmeid modelleerimiskeeli (nt SecureUML, UMLSec, jne) mis esindavad RBAC-i, kuid nad ei ole koosvõimelised (raske selgitada) ning neid ei ole lihtne võrrelda omavahel. Iga modelleerimiskeel esindab erinevaid perspektiive informatsioonisüsteemides. Pealegi on vajadus ühendada disain ja nõudestaadiumid selleks, et avastada süsteemi turvalisusprobleemid ja analüüsida seotud turvalisuskompromisse varasemates staadiumites. KAOS on eesmärgipõhine nõue tehnikavaatenurgast, et paika panna tarkvara nõuded. Sellel hetkel, KAOS on tulevikus võtmelahendus selleks, et kombineerida nõuded disainipõhimõtetega.

Selles teesis me analüüsime KAOS-e võimet kohaneda RBAC-ile. Täpsemalt, me kasutame süstemaatilist lähenemist selleks, et aru saada kuidas KAOS-t on võimalik kasutada nii, et see kohanduks RBAC-ile. Meie uurimistöö põhineb transformatsioonireeglitel KAOS-SecureUML-i ja KAOS-UMLSec-i vahel. Pealegi, läbi nende muutuste näitame me kuidas sobitasime KAOS-e RBAC-ile.

Selle uurimistöö esitamisel on mitmeid kasutegureid. Esiteks, see aitab potentsiaalselt mõista kuidas KAOS toimib koos RBAC-iga. Teiseks, see defineerib lähenemise välja meelitada turvanõuetele IS-i varajastes arendusfaasides RBAC-i jaoks. See rakendab meie tulemused juhtumuuringus selleks, et mõõta määratletud lähenemise õigsust. Kolmandaks, see transformatsioon KAOS-est/KAOS-eni aitaks IS arendajaid ja teistel süsteemi osanikel (nt süsteemianalüütikuid, süsteemi administraatoreid jne) mõista kui tähtsad need turvalisuslähenemised on ja millistel on rohkem eeliseid/puudusi. Me planeerime kehtestada oma tulemused selleks, et reegleid ja modeleid muuta olenevalt nende õigsust, mida mõõdetakse. Viimaseks, me oleme võimelised õigustama oma disainistaadiumit nõudmise staadiumiga.

REFERENCES

[1] Ferraiolo D. F., Sandhu R., Gavrila S., Kuhn D. R. and Chandramouli R., *Proposed NIST Standard for Role-Based Access Control*, ACM Transactions on Information and System Security, Vol. 4, No. 3, August 2001.

[2] Jurjens, J., *Secure Systems Development with UML*. Heidelberg, German, Springer-Verlag, 2004

[3] Kitchenham, B., Pfeeger, S. L., Fenton, N.: Towards a Framework for Software Measurement Validation, IEEE Trans. on Soft. Eng., IEEE Press 21 (12), 1995.

[4] Letier E., Reasoning about Agents in Goal-Oriented Requirements Engineering. PhD thesis, Universit´e Catholique de Louvain, 2001.

[5] Lodderstedt T., Basin D. and Doser J., *SecureUML: A UML-Based Modeling Language for Model-Driven Security*. «UML» 2002: The Unified Modeling Language**:** 426-441, 2002.

[6] Matulevičius R., Heymans P., and Opdahl A. L., *Ontological Analysis of KAOS Using Separation of Reference*, EMMSAD 2006.

[7] Matulevičius R., and Dumas M., *Towards Model Transformation between SecureUML and UMLsec for Role-based Access Control*, 2011.

[8] Matulevičius R., Habra N., and Kamseu F., *Validity of the Documentation Availability Model: Experimental Definition of Quality Interpretation*, 2010.

[9] Nhlabatsi A., Bandara A., Hayashi S., Haley C. B., Jurjens J., Kaiya H., Kubo A., Laney R., Mouratidis H., Nuseibeh B.. Tun T. T., Washizaki H., Yoshioka N., Yu Y., *Security Patterns: Comparing Modeling Approaches*, December 2009.

[10] Respect-IT. A KAOS Tutorial. V1.0. 18 October 2007.

[11] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., Youman, C. E., *Role-Based Access Control Models*, February 1996.

[12] Van Lamsweerde, A., *Requirements Engineering*. From System Goals to UML Models to Software Specifications. West Sussex: Wiley, 2009.

[13] Van Lamsweerde, A., *Elaborating Security Requirements by Construction of Intentional Anti-Models*, Proc. ICSE'04: 26th International Conference on Software Engineering, Edinburgh, ACM-IEEE, May 2004, 148-157.

[14] Van Lamsweerde, A., Darimont, R., Letier, E., *Managing Conflicts in Goal-Driven Requirements Engineering*, IEEE Transactions on Software Engineering, Special Issue on Managing Inconsistency in Software Development, Nov. 1998.

[15] Van Lamsweerde A., The KAOS Meta-model: Ten Years After. Technical report, Universite Catholique de Louvain, 1993.

[16] Van Lamsweerde A., Feather M. S., Fickas S., Finkelstein A., Requirements and Specification Exemplars, *Automated Software Engineering*, 4 (1997) 419–438.

[17] Wikipedia (2012) - Access Control. http://en.wikipedia.org/wiki/Access_control. Last Accessed October 24, 2012.