TÉCNICO LISBOA

Blended Workflow Access Control Perspective

Using Alloy Specifications

Frederico Miguel Castelo Madeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Members of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

June 2018

Acknowledgments

I would like to thank my family and friends for their friendship, encouragement and care over all these years. I would also like to acknowledge my dissertation supervisor Prof. Antonio Rito Silva for his insight, tremendous support, availability and sharing of knowledge that made this Thesis possible. Then I would like to thank to all my friends and colleagues that helped me grow as a person and were always there and with whom i shared great moments. Last but not least I would like to thank my significant other for everything.

To each and every one of you – Thank you.

Abstract

The Blended Workflow is a project that combines different representations for the same workflow specification allowing the end user to follow the standard activity-based view when doing his work, but to change that view to a goal-based when an unexpected situation occurs. This work has the objective to enrich the blended workflow model with the access control perspective. Therefore, in this work i model and validate the access of end users to application resources, in order to guarantee that only proper accesses are made. This validation is achieved using a tool called Alloy Analyzer.

Keywords

Blended Workflow, Access Control, Permission, Alloy, Validation

Resumo

O Blended Workflow é um projeto que combina diferentes representações da mesma especificação do fluxo de trabalho permitindo assim ao utilizador final seguir um caminho baseado em atividades enquanto efetua o seu trabalho, mas ter a opção de alterar para um caminho baseado em objetivos quando situações inesperadas ocorrem. Este trabalho tem como objetivo enriquecer o modelo do Blended Workflow com controlo de acesso. Com esse objetivo, neste trabalho é modelado e validado o controlo de acesso de utilizadores finais aos dados da aplicação de modo a garantir que apenas acessos devidos ocorrem. A validação é obtida através da utilização de uma ferramenta chamada Alloy Analyzer.

Palavras Chave

Blended Workflow, Controlo de Acesso, Permissões, Alloy, Validação

Contents

List of Figures

Listings

1

Introduction

Contents

Nowadays, every organization has computer systems where all resources that in the past centuries existed only in paper format are now in that computer systems, this collection of resources can be numerous things such as: clients information, past transactions details, organization inventory, work schedules and all other types of data. With all that information on computer systems, there was a normal tendency to use that computer systems to everything support the organization's business processes.

A business process is a set of tasks or activities related that as a whole are a mean to achieve the goal or product that the organization wants to achieve. Normally those tasks have an order, for example in the case of an application, those tasks can be, specify, model, develop, test, correct bugs, and deliver to the costumer [2].

With all this business process information in computer systems, a demand of tools to manage that information appeared, thus workflow systems were created. Their primal focus is to facilitate the management of the business process providing an infrastructure to define, manage and monitor the business process. Additionally, they provide the users a better view of their business process work, so the user can easily know what has been made, what is left to do and the current state of the business process. There numerous tools for manage the workflow, normally they only support one view of the workflow specification, but there is a lack of tools that support different views for the same workflow specifications [3].

The Blended Workflow [1] project is an approach that supports four different representations of the same workflow: one data-based, condition-based, goal-based, and activity-based. With the possibility of different representations there are different ways that the workflow instances can execute. The goal and activity-based provide views for the execution of blended workflow instances, the activity view shows the user the sequence of activities or tasks they have to preform to achieve the business goal, on the other hand, the goal view allows the user to proceed in a different way from the normal behavior and sequence of the activity view in order to adapt to unexpected situations that may occur and change the course of the business process.

Workflow management systems tend to use resources, which access must be controlled. A common solution is access control which is composed by a set of permissions that allow users to access only the information that they are authorized in order to prevent unauthorized accesses to sensible information or to information that the the user has no rights to.

Implementing an access control is a very sensitive task due to the fact that the access control is the primary barrier that ensures the security of the data of an application, in order to make a correct implementation first there is a need of a correct specification, model and validation of that model.

The tool used in this work for the purpose of the validation of the model is Alloy Analyzer[1]. This tool uses alloy language which is used for numerous applications such as finding holes in security mecha-

[1] http://alloy.mit.edu/alloy/

nism, Alloy language [4] is based on relations which Alloy uses to represent every types of data such even sets, tuples, scalars. In the Alloy Analyzer models can be made and add constraints that can be solved by the tool, additionally the tool has the ability to generate a graphic visualization for the model by the user defined and to check user defined properties, generate a counterexample and it's correspondent graphic visualization.

1.1 Motivation

In computer systems security is a main concern, and when there is data in an application, access control restrain everyone from directly access that information. The focus here is that all users cant have the same permissions to access a resource that is part if the application data and this is the main challenge of the access control implementations, some data may be only accessed by people who preform specific task, other can only be accessed for some user that has a specific role, or can only be accessed with other user authorization [5]. This set of constraints and restrictions are what define the access control.

With the knowledge that the access control has a very important role in a workflow management system, it is essential that its specification be as correct and accurate as possible in order to prevent possible incoherences and bugs. The specification is one of the most important part of the access control because a wrong specification may lead to breaches in the application security and allow unauthorized access to sensible information, to validate the correct specification all the constraints defined must be evaluated with the use of the Alloy Analyzer. What makes Alloy Analyzer such a great tool is the ability to visualize the partial model of the representation of the specification which makes it easier to represent incrementally the model, the graphical visualization also allows to navigate the model to better understanding of possible specification mistakes and the structure itself, additionally the user can test conditions cases that should occur as stated in the defined specification and Alloy Analyzer has the ability to generate counterexamples if they exist, which allows the user to better understand the faults on the specification.

1.2 Objectives

The Objectives of this work is to define the specification of the access control for blended workflow, prove the correct specification through validations of the defined access control.

2

Blended Workflow

Contents

2.1 Blended Workflow

The Blended Workflow approach [1] was created with the intent of take advantage of both data and artifact-based approaches. Despite using various approach, it is possible to preserve the normal behaviour when executing the business process. The Blended Workflow approach combines an activity-based representations which leads the user to follow a standard behaviour when executing their tasks, with a goal-based representation which allow the user to follow an alternative path to achieve the business goal when an unexpected situation occurs while preserving the same business goals.

The process of designing is composed by four stages, each one of the stages produce a unique representation for the specification of the Blended Workflow: data model, condition model, goal model and activity model as shown in Figure 2.1.

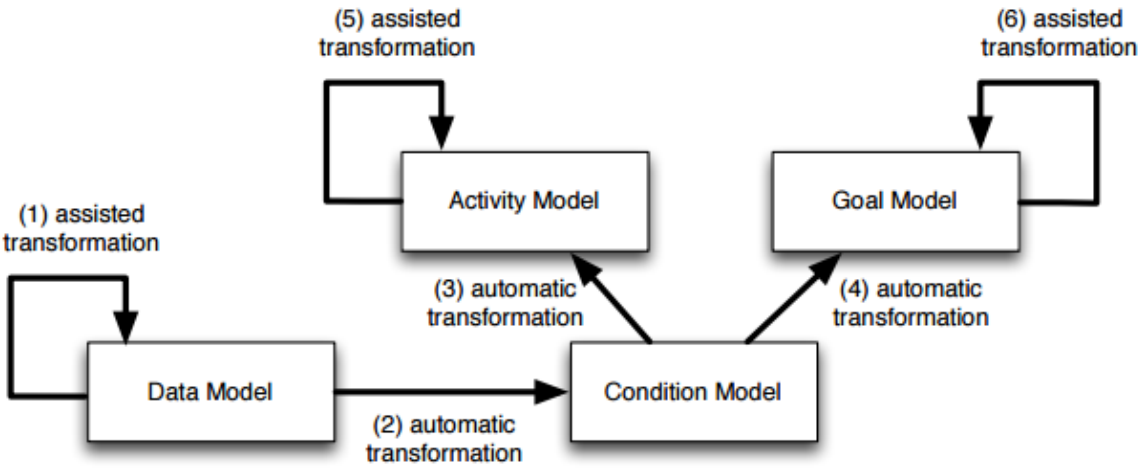

Figure 2.1: Blended Workflow Design Process [1]

These processes based on the data model, which can be modified through the transformation (1). There are three possible operations: add, remove or update over 4 possible resources: entities, attributes, relations and dependencies. These operations do not preserve the models equivalence and every time the data model is changed the other three models are automatically regenerated through transformations (2), (3) and (4). The condition model is generated automatically and is the only one that does not support any transformation, this model contains the conditions regarding the workflow instances. The condition model generates two different models,the activity model through transformation (3) and the goal model through transformation (4). Both activity and goal model support transformations but this transformations preserve the equivalence of all models. In the activity model it is possible to perform the operations: rename, merge and split to an activity and the operations: add and remove to a sequence and in the goal model it is possible to perform operations: rename, merge and split to a goal.

Considering the simple case which contains two entities, EntityA has two attributes and EntityB has one attribute. Between them there is a relation of one-to-many. There is also a dependency of

between attributeThree and attributeTwo, which means that the attributeThree can only be defined if the attributeTwo is already defined. This simple case generate four different models as shown in figure 2.2.

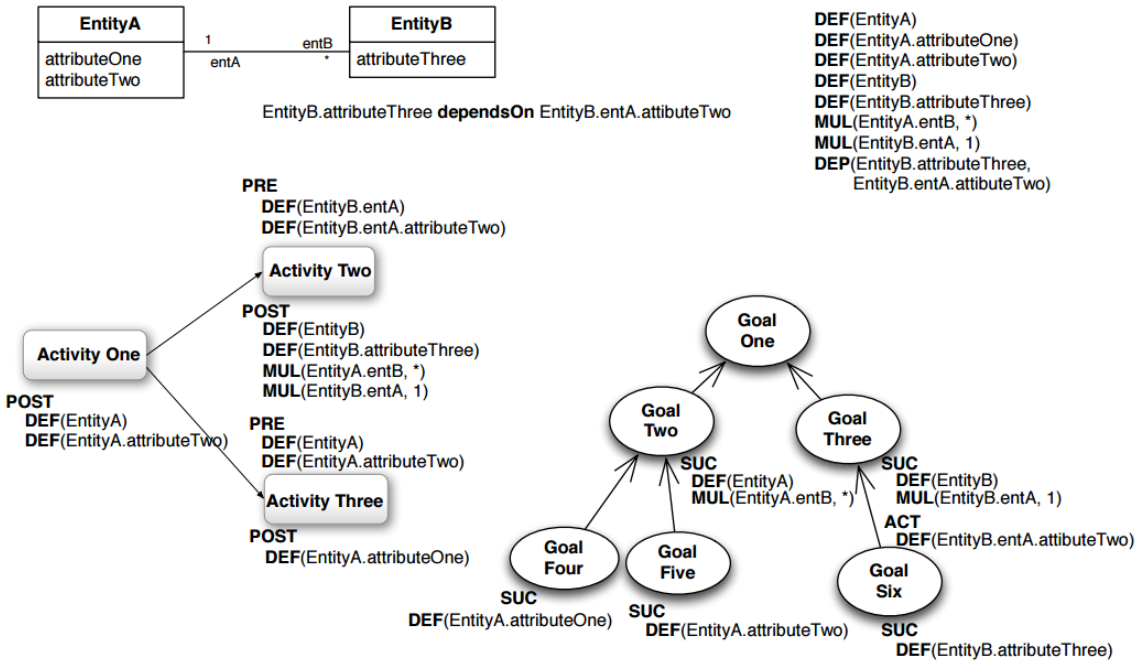

Figure 2.2: Simple Case Models [1]

The top left model is the data model of the simple case which is a diagram class annotated with dependencies. The top right model is the condition model, this model contains all the condition for defining the entities, attributes, relations and dependencies. On the bottom left is the activity model which is a result of transformation (3) and some transformations (5). The pre-conditions, represented above the activity box represent the conditions that need to be achieved in order to start that activity and the post-conditions represented below the activity box represent the conditions that need to be achieved to finish the activity. On the bottom right is the goal model which is a result of transformation (4) and some transformations (6). I this model the goals are activated from the top to the bottom but are achieved from bottom to the top. In this case the EntityA attributes are defined an goal four and five are achieved, the same happens for EntityB attributes and goal six, but because of the attribute dependency goal six can only be achieved after goal five. Then EntityA and his multiplicity are defined and goal two is achieved and the same for EntityB and his multiplicity and goal three is achieved. After goal two and three are achieved, the top goal is also achieved.

3

Related Work

Contents

In this section I discus the related work researched for the purpose of studying the state of art on access control in workflow systems and the use of Alloy to the specification of access control and security. In the following sections three works are presented which provide a base to define the access control of this work. Note that those works do not represent all the research done but the work I considered relevant for the access control perspective of the Blended Workflow.

3.1 Access Control Requirements in Workflow Management Systems

There are many requirements [5] that need to be met in order to achieve a proper access control in a workflow system. In the work produced by Wu, S., Sheth, A. P., Miller, J. A., & Luo, Z. (2002) an example of a workflow management system that manages an hospital is presented. The hospital example is good to demonstrate all requirements due to its complexity where there are numerous possible tasks and constraints to access resources such as patients medical files.

The role is the base requirement, depending on its role an hospital worker can perform certain tasks, for example the receptionist has a role that grants him permission to register a patient in the hospital system. Defining permission based on roles allows to group permissions common to different users in a role. This avoids the repetition of the same permissions for the different users.

Sometimes the role is not sufficient to grant permission to perform a task and, for that reason, it is introduced the concept of process instance based user group, where in addition to the role, the person performing some task must be part of a restrict group. In the hospital scope the role of doctor it is not sufficient to access a patient health care records, it is also necessary that the doctor is part of the team providing health care treatment to that specific patient. This requirement is useful to define rules for particular instances of the resources in design time.

In some situations the permission is based in the role of the subject and the task he is performing, meaning that a subject that has a certain role may have the ability to access certain information only when performing a particular task. For example in the hospital case a pharmacist may dispense medicine for a patient according to the physician description and perform medical consultation to a physician, for the first task, he cannot have access to the patient records, but for the second he needs the permission to access that records to provide accurate consultation.

The three requirements presented refer to the subject performing the task, in those requirements the permission is based on the role, user group, or the task he is performing. Despite this, permissions always apply to resources. The next requirement is related to those resources. Permissions apply to a particular resource, they define the ability to perform some action upon the resource. For example a doctor must have permission to access medical history of its patients. The authors also state that one

of the down sides of the definition of permission to resources is the huge number of existing tuples in the permission database. Knowing that the definition of permissions over resources generates multiple tuples, one way to overcome this obstacle is to define permissions over tasks. Having permission to perform a task the subjects can access all the resources used to execute that task. In the case of the Blended Workflow, it is going to be defined permissions over activities and goals and the subject will have permission to access all resources in the conditions of the activities and goals.

Another requirement is the privilege propagation, in this case the permissions a subject has, may be propagated to another subject in some cases. As examples, when a the patient's doctor needs a second opinion from another doctor his permissions to access the patient records should be propagated to the doctor providing that opinion. This requirement is also useful when the permissions of two users are not sufficient to perform a task individually, but together they have the permissions to perform the task.

The last requirement is the dynamic authorization, which refers to all cases where the permissions are defined or changed in execution time. The majority of the permissions are defined in design time before the beginning of the task execution but there must be some cases where during the execution the permissions may change, for example, when a doctor replaces another he should inherit the privileges of its predecessor or when a doctor wants to access certain information of the patient records that can only be done with the patient authorization. This last requirement it is the most difficult to specify due to alterations in the access control rules when certain events occur.

3.2 Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns

The authors Joseph P. Near and Daniel Jackson proposed a technique to find bugs in applications using a catalog of access control patterns [6], this catalog was based on role-based access control and explored seven of the most common mistakes made when developing a web application. This work focus on the fifty most watched Ruby on Rails applications on GitHub. The authors used symbolic execution to extract unauthorized accesses from the applications source code.

The first security pattern is the ownership which refers to the relations between a subject and the resource created by him. The creator of said resource should automatically have read and right permission over that resource. In order to represent this pattern the authors defined a relation between subject and the resource owned by him. This relation allows the user to access the resource without the need of explicit permissions.

The second pattern is related to public objects, when available everyone should have access to read them even if their no authenticated. This public objects are an extension of objects that do not need permissions to be read. Although everyone can access them even without authentication, only subjects

with permissions can define or modify them. This is very common in web applications such as blogs and news websites where articles can be available to without the need of authentication.

The third pattern is related to authentication. In every application with access control there is an authentication method and one common mistake developers make is forgetting to check if a user is logged in before allowing an operation. A user may have permissions to perform an operation and not being logged in. With that in mind, for all non public resources the user must be authenticated and logged in the application in order to access any resource.

The fourth pattern is the explicit permissions. This pattern is common to most web application, and states that the permission should be explicitly stored in the application as resources. This allows the verification of permission whenever a subject tries to access a resource. The application will search through its permission resources and ensure that permissions that allow the subject to perform an operation exist. This happens in applications where by default no one has permission over a resources unless there is a permission resource that states otherwise or the resources are public.

The fifth pattern is related to users profiles which exist in numerous web applications, and a common mistake the developers make is to forget to check if a user is the owner of that profile before updating it, this pattern ensures that only the owner of the profile can write in that profile in order to avoid disruptive modifications to users profiles.

The sixth pattern is the administrator. The administrator is normally a special class of user that can perform every operation over every resource. The authors propose a role to specify the administrator which have read and write access to all resources. In the case of the Blended Workflow and considering that the permissions can be specified over operations such as activities and goals instead of resources, the roles should also have permissions to execute those operations.

The last pattern is related to role and to the fact that in some cases applications define various roles and represent them explicitly using a resource, these roles are assign to users and the permissions are granted based on these roles. This allows to create permissions where the subject is not a user but instead it is a role. This also allow the definition of permissions without having any user in the domain. When the users are added to the application there should be assigned roles to the users.

With this catalog of security patterns and the mapping of the application types to the Alloy types, the authors achieved aa effective method to automatically test applications for bugs that can scale easily, produces few false positives and was able to find previously unknown bugs..

3.3 Enabling Fine-grained Access Control in Flexible Distributed Object-aware Process Management Systems

Kevin Andrews, Sebastian Steinau, and Manfred Reichert propose a access control in order to increase the flexibility of the distributed object-aware process management systems [7]. The access control presented by the authors is part of the PHILharmonicFlows framework [8]. This framework provides a distributed data-driven process execution engine. The access control is role based, and contains seven requirements.

The first requirement states that in order to activate permissions and roles dynamically at run-time the access control system must use concrete elements of the domain represented. In the framework of this work the authors use the concepts of the role-based access control like users, roles, permissions, operations and objects. Although the access control uses those same concepts, it is not sufficient to ensure flexibility for large corporation, where there are a big influx of workers and others that switch to different jobs that require different permissions. This causes a many changes in the permissions and roles of each employee leading to a difficult task for the administrative that manages permissions. With this requirement a user can only activate a role which is assigned to him and when performing certain task. For example the employee of a bank needs to change the balance of a checking account. There is a permission that maps a role to the permission to change the balance of that account. In order to do so the employee should be authorized to activate that role. This authorization to activate roles is represented through constraints which depend of the scope of the domain and are defined when assigning roles.

The second requirement states that the role authorization can be defined based on the attributes of a user. The users are represented by objects in the domain and, in order to authorize roles in run-time, the users attributes must be used. These attributes are one of the two factors that differentiate users instances. For example, an employee object which attribute department has the value of account management should be able to authorize the account manager role.

The third requirement, is similar to the previous, but instead of authorizing role activation based on the user attributes, it authorizes based on domain relations. These relations are the second factor that distinguishes instances of users in the domain. For example, an employee that is associated with a costumer that has a checking account, should be able to activate the role checking account manager in order to perform operations in that costumer checking account.

The fourth requirement is the permission activation depending on resource state. This requirement consider that the user has the role activated to perform an operation to a certain resource but can only perform that operation if the resource is in certain state. For example, an employee that have the checking account manager role active in respect to a transfer resource, besides approving the transfer,

he may also activate the permission to write a comment attribute when the transfer is in decision state. This comment attribute is only relevant in this state and should not be written in another state of the transfer.

The fifth requirement is related to the resources data and only allows the user to activate a permission based on the value of an attribute of the resource. This requirement also considers the case where a user already has the role activated. For example, an employee with an accounting manager role active regarding a transfer can only approve the transfer if the attribute amount is less than 50.000€. Otherwise, it is necessary to have the role supervisor active.

The sixth requirement is scalable real time permissions and role authorization. The permissions and role authorizations are granted and also revoked during the execution and it is essential that at any time the access control does not grant outdated permissions. For example, a costumer that edits the amount of the transfer to over 50.000€, should force a revoke of the permission to authorize the transfer to an employee with a role of checking account manager active. This modification occurs in real time in the domain and should be known to all participants immediately.

The last requirement is adequate performance and caching results. The authorization of permissions and roles should not be based in cached information in order to ensure performance but by other means that make the previous requirement possible but at the same time ensures adequate performance.

4

Context

Contents

The access control specification is achieved through the enrichment of the Alloy specification of the Blended Workflow. Currently the specification is divided in two parts: the generic to all models and the specific to each model. In this chapter the Blended Workflow specification is described to provide a proper context for a better comprehension of this work.

4.1 Generic

The generic part contains the Alloy specification that does not change independently of the specific model. In here there is the Blended Workflow specification which represents the structure for the three models. Additionally there are three specification which represent the conditions of each model.

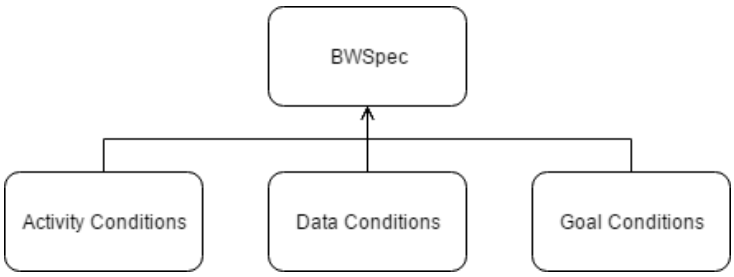

Figure 4.1: Blended Workflow Generic Structure

4.1.1 Generic Blended Workflow Specification

The `BWSPec` on the Figure 4.1 represents the generic Blended Workflow specification. In order to define the Blended Workflow it is necessary to specify the state, as shown in Listing 4.1, which is used to represent the execution of the Blended Workflow . The state is represented by the signature `AbstractState`, which is used in the three models. Each state has the resources already created. In order to represent the resources, this specification introduces two concepts. These concepts are: `Obj` which represents an entity and `FName` which represents an attribute or association between entities. Although association and attributes are both represented as `FName` they can easily be distinguished in the state. The state has objects and fields, the objects represent the entities through a set of signatures of the type Obj and the fields represent the attributes and associations through a function that maps `Obj` x `FName` x `DefVal` and `Obj` x `FName` x `Obj` respectively. In the Blended Workflow models it is only relevant to represent whether the attribute is defined being irrelevant what is the concrete value. Additionally there may exist dependencies between resources. These dependencies are defined as signatures which contains `sourceObj`, `sourceAtt`, `sequence`, `targetAtt`. Although there are four field it is possible that some of them have no value.

Listing 4.1: Generic Blended Workflow Specification

```
1 abstract sig AbstractState {
2     objects: set Obj,
3     fields: objects -> ( FName ->set { Obj + Val } ) ,
4 }
```

4.1.1.1 State Transition Invariants and Completion Rules

Although the invariants and rules are only used in the specific part, their definition is generic. The state transition invariants is the set of invariants that must be ensured whenever a transition between states occur. The completion rules a a set of rules that must be achieved in order to obtain a meaningful complete state, the workflow finishes. In the states transition invariants there are four invariants that must be ensured. The first one is the `noExtraFields` which ensures that an Entity only has as fields its attributes or associations and noting else. The secound one is the `noMultiplicityExceed` which is used during the execution to ensure that the maximum multiplicity is always respected. Then there is the `bidirectionalPreservation` which ensures that during the execution process either an association is bidirectional and each `Obj` instance of the association is in the fields of the other or it is unidirectional and it is possible to create the inverse link of the association in order to make the association bidirectional. The last invariant is used to represent the dependencies. This invariant has a source object, a source attribute, a path form the source object to the target attribute object and a target attribute. Whenever a dependence exists the target resource and path from the source to the target must already be defined in order to define the source of the dependence. The dependence. When the dependencies do not have source attribute nor target attribute it is a dependence between two objects, if there is no source object it is a dependence between an object and an attribute, if there is no target attribute it is a dependence between an attribute and an object and if there is no path then it is a dependence between two attributes of the same object.

The completion rules are composed by four rules. The first, called `attributesDefined`, ensures that for each entity their attributes must be defined to be consider complete. The second rule is the `multiplicityRule` that guarantee the that multiplicities restrictions, both min and max, are respected. The third one is the `bidirectionalRule` which ensures that the associations are bidirectional and each `Obj` instance of the association is in the fields of the other which is needed to achieve a meaningful complete state. The last one is the same representation of the dependencies that appear in the state transition invariants.

Listing 4.2: Generic Invariants and rules

```
1 pred noExtraFields(s: AbstractState, objs: set Obj, fNames: set FName) {
```

```
2       all obj: objs <: s.objects |no obj.(s.fields)[ FName - fNames ]
3   }
4
5   pred attributesDefined(s: AbstractState, objs: set Obj, atts: set FName) {
6       all obj: objs <: s.objects | all att: atts | s.fields[obj, att] = DefVal
7   }
8
9   pred multiplicityRule(s: AbstractState, objs: set Obj, role: FName) {
10      all obj: objs <: s.objects
11          | #s.fields[obj, role] >= role.minMul and #s.fields[obj, role] <= role.maxMul
12  }
13
14  pred noMultiplicityExceed(s: AbstractState, objs: set Obj, role: FName) {
15      all obj: objs <: s.objects | #s.fields[obj, role] <= role.maxMul
16  }
17
18  pred bidirectionalRule(s: AbstractState, objsOne: set Obj, roleOne: FName, objsTwo: set Obj,
19                         roleTwo: FName) {
20      all objOne: objsOne <: s.objects
21          | all objTwo: s.fields[objOne, roleTwo] | objOne in s.fields[objTwo, roleOne]
22      all objTwo: objsTwo <: s.objects
23          | all objOne: s.fields[objTwo, roleOne] | objTwo in s.fields[objOne, roleTwo]
24  }
25
26  pred bidirectionalPreservation(s: AbstractState, objsOne: set Obj, roleOne: FName, objsTwo: set Obj,
27                                 roleTwo: FName) {
28      all objOne: objsOne <: s.objects | all objTwo: s.fields[objOne, roleTwo]
29          | objOne in s.fields[objTwo, roleOne] or canLink[s, objTwo, roleTwo, objOne]
30      all objTwo: objsTwo <: s.objects | all objOne: s.fields[objTwo, roleOne]
31          | objTwo in s.fields[objOne, roleTwo] or canLink[s, objOne, roleOne, objTwo]
32  }
33
34  pred checkDependence(s: AbstractState, sourceObj: Obj, dependence: Dependence) {
35      (dependence.sourceAtt = none and dependence.targetAtt = none) implies {
36          all oS: sourceObj <: s.objects | !no reach[s, oS, dependence.sequence]
37      } else (dependence.sourceAtt = none) implies {
38          all oS: sourceObj <: s.objects
39              | DefVal in s.fields[reach[s, oS, dependence.sequence], dependence.targetAtt]
40      } else (dependence.targetAtt = none) implies {
41          all oS: sourceObj <: s.objects
42              | (s.fields[oS, dependence.sourceAtt] = DefVal) implies !no reach[s, oS, dependence.sequence]
43      } else (dependence.sequence = none -> none) implies {
44          all oS: sourceObj <: s.objects | (s.fields[oS, dependence.sourceAtt] = DefVal)
45              implies DefVal in s.fields[oS, dependence.targetAtt]
46      } else {
47          all oS: sourceObj <: s.objects | (s.fields[oS, dependence.sourceAtt] = DefVal)
48              implies DefVal in s.fields[reach[s, oS, dependence.sequence], dependence.targetAtt]
49      }
50  }
```

4.1.1.2 Models Generic Predicates

Before explain each of the models, it is necessary to explain some predicates that are used in all models. These predicates are used in the conditions that cause transition between states. These predicates are `noFieldChangeExcept` and `canLink`. The `noFieldChangeExcept` that ensures all fields except the one that is being modified keep the same value. The `canLink` predicate ensures that either both objects did not achieved the maximum multiplicity of association or the source object did not achieved the maximum multiplicity and the target object is already associated to the source object.

Listing 4.3: Models Generic Predicates

```
1  pred noFieldChangeExcept(s, s': AbstractState, asg: set Obj ->FName) {
2      all obj: s.objects - asg.FName | obj.(s'.fields) = obj.(s.fields)
3      all o: asg.FName | all field: atts[s, o] - o.asg | s'.fields[o, field] = s.fields[o, field]
4  }
5
6  pred canLink(s: AbstractState, objSource: Obj, roleSource: FName, objTarget: Obj) {
7      // source is not completely commited yet in the number of targets
8      let commitedTargetObjects = commitedAssociatedObjects[s, objSource, roleSource] |
9          #commitedTargetObjects < roleSource.inverse.maxMul or objTarget in commitedTargetObjects
10     // target is not completely commited yet in the number of sources
11     let commitedSourceObjects = commitedAssociatedObjects[s, objTarget, roleSource.inverse] |
12         #commitedSourceObjects < roleSource.maxMul or objSource in commitedSourceObjects
13 }
```

4.1.2 Data Conditions

In the data model there are three operations, as shown in Listing 4.4, that are the condition which cause a transition between states. These operations are represented as predicates. They are the creation of an entity, an attribute and an association between objects represented respectively as `defObj`, `defAtt`, and `linkObj`. In order to create an entity, it is necessary that the entity does not exists already in the model. For the definition of an entity attribute it is necessary that the entity already exists and that the same entity does not have already that attribute defined. Additionally the predicate `noFieldChangeExcept` is used to ensures all fields except the one that is being modified keep the same value. In both attribute and entity definition, the resource that is being defined may have a dependence, whenever that occurs the resources on which it depends must already be defined. The relation between two objects is unidirectional and to ensure that all relations are bidirectional there is the need to execute the operation two times, for example when trying to link object A and B, the operation link is executed a first time where A is the source object and B is the target object and a second time where B is the source object and A is the target object. During the execution of the operation both objects must exist in the model, and the target object should not be already assigned to the source object. in order to ensure that the operation

can be done properly it is used the `canLink` predicate. Like in the `defAtt` operation, in the `linkobj` the predicate `noFieldChangeExcept` is present to ensure that there are no changes in fields except the one that represents the relation.

Listing 4.4: Data Model Operations

```
1  pred defObj(s, s' : AbstractState, o: Obj) {
2      o !in s.objects
3      s'.objects = s.objects + o
4      s'.fields = s.fields
5      all dep: Dependence | o in dep.sourceObj implies checkDependence[ s', o, dep]
6  }
7
8  pred defAtt(s, s': AbstractState, o: Obj, att: FName) {
9      o in s.objects
10     no s.fields[o, att]
11     s'.objects = s.objects
12     s'.fields = s.fields + (o -> att -> DefVal)
13     noFieldChangeExcept[s, s', o -> att]
14     all dep: Dependence | o in dep.sourceObj and att in dep.sourceAtt implies checkDependence[ s', o, dep]
15 }
16
17 pred linkObj(s, s': AbstractState, objSource: Obj, roleSource: FName, objTarget: Obj, roleTarget: FName) {
18     objSource in s.objects
19     objTarget in s.objects
20     objTarget !in s.fields[objSource, roleTarget]
21     canLink [s, objSource, roleSource, objTarget]
22     s'.objects = s.objects
23     s'.fields = s.fields + (objSource -> roleTarget -> objTarget)
24     noFieldChangeExcept[s, s', objSource -> roleTarget]
25 }
```

4.1.3 Activity Conditions

The activity model uses the same state as the data model but it may contain more complex operations for state transition. An operation associated to a state has two predicates which represent their pre-conditions and post-conditions. The pre-condition predicate has as parameters the entities and attributes that must be defined in order to initiate the activity. Although there are two parameters the predicate may receive only entities, only attributes or none of them. The post-condition predicate has as parameters every resource that will be created with the execution of the activity. The post-condition has three parameters which represent each of the three resource types. If there are entities in the post condition, they cannot exist yet in the model. For the attributes in the post-condition, their entity instances either were already defined or are being also defined in the post-condition. The same happens for the associations in the post-condition, they can only be created if both source and target objects instances

are already in the model or are defined in the post-condition.

Listing 4.5: Activity Model Representation

```
pred preCondition(s: AbstractState, entDefs: set Obj, attDefs: set Obj -> FName) {
    entDefs in s.objects
    attDefs.FName in s.objects
    all obj: attDefs.FName, field: obj.attDefs
        | s.fields[obj, field] = DefVal or s.fields[obj, field] in s.objects
}

pred postCondition(s, s': AbstractState, entDefs: set Obj, attDefs: set Obj -> FName,
                   muls: set Obj -> FName -> Obj) {
    (entDefs != none) implies {
        entDefs !in s.objects
        all obj: attDefs.FName | obj in s.objects + entDefs
        all obj: attDefs.FName, role: obj.attDefs | no s.fields[obj, role]
    }
    all objSource: (muls.Obj).FName, objTarget: FName.(objSource.muls),
        roleTarget: objSource.muls.objTarget | canLink[s, objSource, roleTarget.inverse, objTarget]
    s'.objects = s.objects + entDefs
    s'.fields = s.fields + attDefs  -> DefVal + muls
    noFieldChangeExcept[s, s', attDefs + muls.Obj]
    all o: entDefs, dep: Dependence | o in dep.sourceObj implies checkDependence[ s', o, dep]
    all o: attDefs.FName, dep: Dependence | o in dep.sourceObj implies checkDependence[ s', o, dep]
}
```

4.1.4 Goal Conditions

The goal model also uses the same state as the activity model and data model. Like the activity model an operation also has two predicates that represent two model conditions. One is the activation condition which represent the resources that must be defined in order to activate the goal, and other is the success condition which represent the resources that must be created in order to achieve the goal.

The activation condition predicate has the resources that must already be created in order to activate the goal as parameters. The entities in the activation condition only need to exist in current state. The attributes must have their entities already created and must exist either as a attribute that maps a `DefVal` or an entity that already exists in the current state. In other hand the success condition entities can not exist in the current model and must be possible to define the attributes in the already existing entities or in the entities that are in the post condition. For the associations in the success conditions it must be possible to associate the source and target entities. The execution of the operation adds the entities, attributes and association to the model and ensure that other already existing fields do not suffer changes.

4.2 Specific

The specific part contains all Alloy specification that is specific to each Blended Workflow specification. It contains the Blended Workflow specification which has the model representation, a meaningful complete state, and the invariants to ensure a proper model construction until it achieves the complete state. There is a specification of the model conditions according to the Blended Workflow representation and a representation of the execution of that conditions for each of the models.

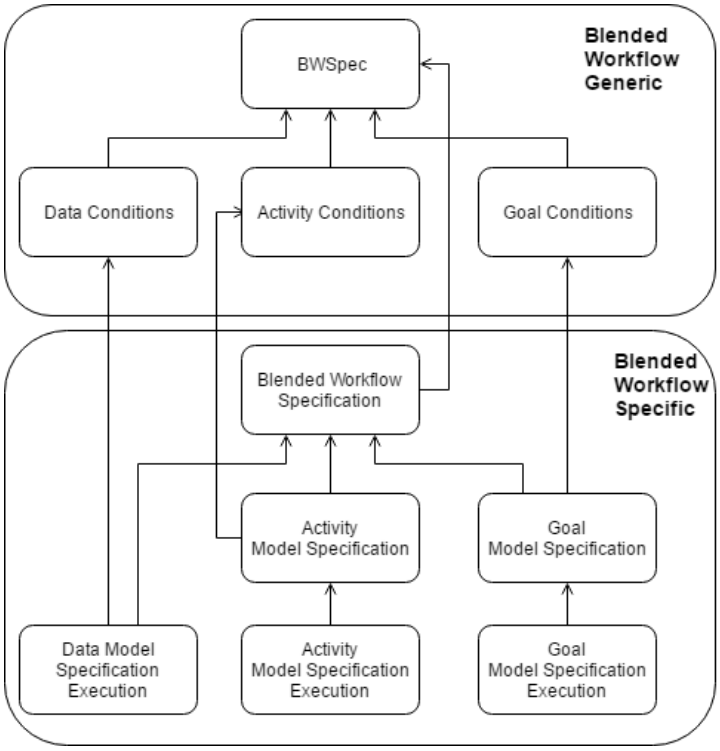

Figure 4.2: Blended Workflow Specific Structure

4.2.1 Blended Workflow Specification

As said in the beginning of this section the Alloy specification is based on a specific Blended Workflow specification. Using a simple example, as shown in Figure 4.3, where there are two entities one with two attributes and other with one and there is an association between them.

The Alloy specification is created based on the model shown in Figure 4.3. In this specification all entities are represented as extensions of objects and the attributes and association are represented as extensions of `FName`. The `FName` which represent an association have inverse, minimum and maximum multiplicity defined. The inverse represents the inverse link between the two entities. Every entity should have a unique name, the same should happen in the fields where are represented both associations

and attributes.

Listing 4.6: Example Model Representation labellst:bwspec

```
sig Patient extends Obj {}
one sig patient_name extends FName {}
one sig patient_address extends FName {}
one sig patient_episode extends FName {}

sig Episode extends Obj {}
one sig episode_reserve_date extends FName {}
one sig episode_patient extends FName {}

sig reserve_date_dependence extends Dependence {}

fact relations {
    // relation Patient - Episode
    patient_episode.minMul = 0
    patient_episode.maxMul = 10
    patient_episode.inverse = episode_patient

    episode_patient.minMul = 1
    episode_patient.maxMul = 1
    episode_patient.inverse = patient_episode
}

fact dependencies {
    // dependence episode_reserve_date from patient_address
    reserve_date_dependence.sourceObj = Episode
    reserve_date_dependence.sourceAtt = episode_reserve_date
    reserve_date_dependence.sequence =  0 -> episode_patient
    reserve_date_dependence.targetAtt = patient_address
}
```

In addition to the resources, the Blended Workflow specifications also contains a predicate that represents a complete state and some conditions that must be met in order to achieved it. This predicate contains the total number of entities instances in the model as well as the total number of each entity type. It also contains predicates that verify that the attribute values are defined, the multiplicity is respected an the bidirectionality of each association is ensured. It is also defined the existent depen-

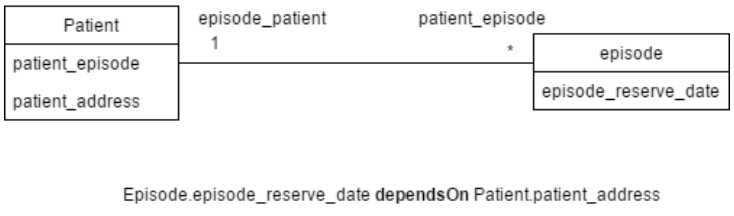

Figure 4.3: Blended Workflow Example

dencies between model resources.

Listing 4.7: Example Complete State

```
pred complete {
    one s: State |
        // cannot be the initial state to find one meaningful state
        #Patient <: s.objects = 1 and
        #Episode <: s.objects = 1 and
        #s.objects = 2 and
        // model is well defined

        // all attributes are defined
        attributesDefined [s, Patient, patient_name + patient_address]  and
        attributesDefined [s, Episode, episode_reserve_date] and

        // associations multiplicity
        multiplicityRule [s, Episode, episode_patient] and
        multiplicityRule [s, Patient, patient_episode] and

        // bidirectional relation
        bidirectionalRule [s, Patient, episode_patient, Episode, patient_episode] and

        // dependencies hold
        checkDependence [s, Episode, reserve_date_dependence]
}
```

4.2.1.1 Invariants

The invariants are the rules that must be preserved during the execution of the Blended Workflow. These invariants are created using the generic predicates of the state transition invariants. This invariants ensure that there are no extra fields except the defined in the Blended Workflow Specific. The maximum multiplicity of an association should not be exceeded. While it is being created there may be states where the minimum multiplicity is not respected, due to its construction process until achieve the complete state where the minimum multiplicity must be respected. Every existing association must be either bidirectional or is unidirectional and it is possible to create the inverse link. Additionally it is ensured that the dependencies are respected.

Listing 4.8: Example Invariants

```
pred Invariants(s: State) {
    // no extra fields
    noExtraFields [s, Patient, patient_name + patient_address + patient_episode]
    noExtraFields [s, Episode, episode_reserve_date + episode_patient]

```

```
6       // does not exceeds mutliplicity
7       noMultiplicityExceed [s, Episode, episode_patient]
8       noMultiplicityExceed [s, Patient, patient_episode]
9
10      // if there is a link between two objects, either is unidirectional or bidirectional
11      bidirectionalPreservation [s, Patient, episode_patient, Episode, patient_episode]
12
13      // dependencies hold
14      checkDependence [s, Episode, reserve_date_dependence]
15  }
```

4.2.2 Data Model Specification and Execution

In the specification of the data model the possible transitions between states are supposed to be defined, instead the operations specified in the data model conditions are used, therefore the data model transitions does not need be specified for each Blended Workflow specification. In order to represent the execution there is the necessity of creating a signature `State` that extends `AbstractState` to create instances that will be used in the execution. Then it is ensured that the first state has no resources and that the only way to transition between states is performing one of the three possible operations of the data model. The definition of an attribute cannot be done without the attribute object already defined and a relation cannot be defined without both objects defined and the possibility to define the inverse relation with the same instances. To represent a proper execution, it is ensured that a complete state can be achieved based on what is defined in the achieve file. Additionally it is tested through the use of asserts that every operation performed in all possible scenarios maintain the invariants stated in the invariants file.

Listing 4.9: Data Model Specification

```
1   sig State extends AbstractState {}
2
3   pred init (s: State) {
4       no s.objects
5       no s.fields
6   }
7
8   fact traces {
9       first.init
10      all s: State - last | let s' = s.next |
11      some p: Patient, e: Episode |
12          defObj [s, s', p] or
13          defAtt [s, s', p, patient_name] or
14          defAtt [s, s', p, patient_address] or
15          defObj [s, s', e] or
16          defAtt [s, s', e, episode_reserve_date] or
```

```
17          linkObj [s, s', p,  episode_patient, e, patient_episode] or
18          linkObj [s, s', e, patient_episode, p, episode_patient]
19  }
20
21  run complete for 4 but 8 State, 5 Int
22
23  assert ExecPreservesInv {
24      all s, s': State |
25          Invariants [s] => Invariants [s']
26  }
27  check ExecPreservesInv for 4 but 8 State, 5 Int
```

4.2.3 Specific Activity Model

4.2.3.1 Activity Model Specification

Contrary to the data model the activity model need a specification for each possible transition between states. These transition are represented using the predicates `preCondition` and `postCondition` specified in the activity model conditions. In this particular case, shown in Listing 4.10, there are three activities which are `registerPatient`, `registerPatientAddress`, `createEpisode` and `bookAppointment`. The `registerPatient` has no pre-condition and has the `Patient` entity and the attribute `patient_name` in the post-condition. The `registerPatientAddress` has the `Patient` as pre-condition and has `patient_address` as post-condition. The `createEpisode` has the `Patient` as pre-condition and the `Episode` and the association between the `Patient` and the `Episode` as post-condition. The `bookAppointment` has the `Episode` as pre-condition and the attribute `episode_reserve_date` as post-condition.

Listing 4.10: Blended Workflow Specific Activity Model

```
1   pred registerPatient(s, s':AbstractState, p: Patient) {
2       preCondition[s, none, none -> none]
3       postCondition[s, s', p, p -> patient_name, none -> none -> none]
4   }
5
6   pred registerPatientAddress(s, s':AbstractState, p: Patient) {
7       preCondition[s, p, none -> none]
8       postCondition[s, s', none, p -> patient_address, none -> none -> none]
9   }
10
11  pred createEpisode(s, s': AbstractState, p: Patient, e: Episode) {
12      preCondition[s, p, none -> none]
13      postCondition[s, s', e, none -> none, (p -> patient_episode -> e) + (e -> episode_patient -> p)]
14  }
15
16  pred bookAppointment(s, s': AbstractState, e: Episode) {
17      preCondition[s, e, none -> none]
18      postCondition[s, s', none , e -> episode_reserve_date, none -> none -> none]}
```

4.2.3.2 Activity Model Specification Execution

The representation of the execution of the design Blended Workflow activity model is where the `State` signature is defined. Like the data model, the activity model has a fact that ensures that the first state does not have any resource and that the state will only transition when one of the activities occur. Besides the representation of the execution of the activities defined in the previous section, there is an assertion that ensures that for every state and any activity that occurred the invariants are always respected.

Listing 4.11: Activity Model Specification Execution

```
1  sig State extends AbstractState {}
2
3  pred init (s: State) {
4      no s.objects
5      no s.fields
6  }
7
8  fact traces {
9      first.init
10     all s: State - last | let s' = s.next |
11     some p: Patient, e: Episode |
12         registerPatient[s, s', p] or
13         registerPatientAddress[s, s', p] or
14         createEpisode[s, s', p, e] or
15         bookAppointment[s, s', e]
16 }
17
18
19 run complete for 4 but 5 State, 5 Int
20
21 assert ExecPreservesInv {
22     all s, s': State |
23         Invariants [s] => Invariants [s']
24 }
25 check ExecPreservesInv for 4 but 15 State, 5 Int
```

4.2.4 Specific Goal Model

4.2.4.1 Goal Model Specification

Like the activity model the goal model need a specification for each possible transition between states. These transition are represented using the predicates `actCondition` and `sucCondition` specified in the goal model conditions. In this particular case, shown in Listing 4.12, there are three activities which are `registerPatient`, `createEpisode` and `bookAppointment`. The `registerPatient` does not have

activation conditions and has the `Patient` and its attributes as success condition. The `createEpisode` has the `Patient` as activation condition and the `Episode` and the association between the `Patient` and the `Episode` as success condition. The `bookAppointment` has the `Episode` as activation condition and the `episode_reserve_date` as success condition.

Listing 4.12: Blended Workflow Specific Goal Model

```
pred registerPatient(s, s': AbstractState, p: Patient) {
    actCondition[s, none, none -> none]
    sucCondition[s, s', p, p -> patient_name + p -> patient_address, none -> none -> none]
}

pred createEpisode(s, s': AbstractState, p: Patient, e: Episode) {
    actCondition[s, p, none -> none]
    sucCondition[s, s', e, none -> none, (p -> patient_episode -> e) + (e -> episode_patient -> p)]
}

pred bookAppointment(s, s': AbstractState, e: Episode) {
    actCondition[s, e, none -> none]
    sucCondition[s, s', none, e ->episode_reserve_date, none -> none -> none]
}
```

4.2.4.2 Goal Model Specification Execution

In the goal model the execution operations are represented as goals. Like in the data and activity models it is defined the signature of `State` which is an extension of `AbstractState` to be used in the execution of the model. In order to represent the execution of the goal model there is a fact that ensures that the first state has no resource and the transition between states can only occur whenever a one of the represented goals occur. Besides the representation of the execution of the activities defined in the previous section, there is an assertion that ensures that for every state and any goal that occurred the invariants are always respected.

Listing 4.13: Blended Workflow Specific Goal Model Execution

```
sig State extends AbstractState {}

pred init (s: State) {
    no s.objects
    no s.fields
}

fact traces {
    first.init
    all s: State - last | let s' = s.next |
```

```
11      some p: Patient, e: Episode |
12          registerPatient[s, s', p] or
13          createEpisode[s, s', p, e] or
14          bookAppointment[s, s', e]
15  }
16
17  run complete for 4 but 4 State, 5 Int
18
19  assert ExecPreservesInv {
20      all s, s': State |
21          Invariants [s] => Invariants [s']
22  }
23  check ExecPreservesInv for 4 but 4 State, 5 Int
```

5

Approach

Contents

In this chapter it is described the objectives of this work and the approach taken in order to achieve them. In the section objectives the access control is briefly described. In the access control approach section it is presented the base structure of the access control, which is common to all security patterns, and its Alloy representation.

5.1 Objectives

The final result of this work is the formal specification of several access control patterns to the Blended Workflow models. The access control proposed is a based on the permissions that a subject have. This subject can have many representations which are presented in their own security patterns. The targets of the base access control being used are the Blended Workflow resources which are entities, attributes and relations. The access control uses several concepts: subject, permission, right and resource. The subject represents is representation of the system user. A permission maps a right that a subject have over a resource, for example, given an object, there are two rights, one to read it and another to define it. In the base access control model, the permissions of the access control will only be over the resource types.

In the data model, which is represented as a class diagram, the access control focus on the operations made to all the resources. Therefore the data model should be enriched to support the association of permissions to each resource in order to define the users permissions. When defining an entity the user needs to have a define permission to the entity type of the instances he can define in order to create one instance of that entity. In order to create attributes the user must have the permission to define that attribute type. The relation needs definition permission over both attributes that represent both directions of the relation. There are some cases where a resource may depend of another, in those cases the user must have all the permissions needed to define that type of resource and must be allowed to read the resource type on which it depends.

In the activity model the resources are grouped in activities. In order to perform an activity a user must have permission to read all the resources in the pre-condition. Additionally for each of the post-condition resources the user must have the same permissions it needs to define each resource in the data model.

The goal model is similar to the activity but groups the resources in goals. A user can achieve a goal in which he has read permission to all resources types in the activation conditions. Additionally for each resource in the success condition the user must have the same permissions it needs to define each resource in the data model.

In addition to the formal specification, a validation is also made for the access control on all models. This validation has two different components. The first one ensures that a meaningful complete state

can be achieved with the model enriched with the access control. The other ensures that for every step of the model it preserves both structure invariants and access control invariants. These access control invariants must ensure that in every state transition the user that performed that operation had the permission required to do so. This will be achieved through the use of a log that will record every operation made.

5.2 Base Access Control Approach

The approach taken to create the specification of the base access control is focused in the already existing Alloy specification of the Blended Workflow. This specification is enriched with the access control formal specification which is divided in two parts: the generic and the specific. Note that both activity and the goal model have a similar specification and so their access control. For that reason only the activity model is described from here forward.

5.2.1 Generic

In generic part the base access control contains three new specifications, one for each of the Blended Workflow models and they contain their enrichment with the access control pattern specification. This has a similar structure to the Blended Workflow current specification in which the data, activity and goal models are associated to the Blended Workflow Specification. The three new specifications use the same secure state which is an extension of the Blended Workflow Specification `AbstractState`.

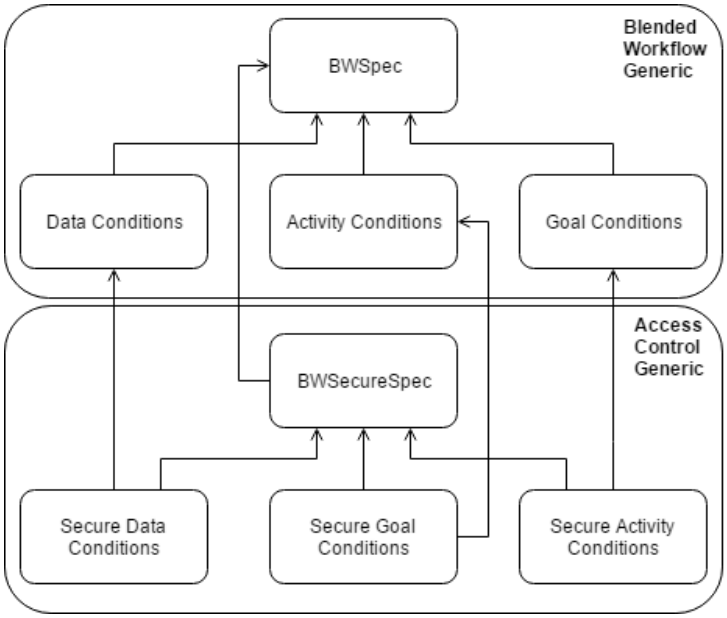

Figure 5.1: Access Control Generic Alloy Structure

5.2.1.1 Blended Workflow Secure Specification

The `BWSecureSpec` enriches the already existing specification with the access control specifics as shown in Listing 5.1. These specifics are `Subject`, `Transition`, `Rights`, `User` and `AccessControlRules`. The `User` is the representation of the person that executes the operation in the Blended Workflow. The `AccessControlRules` is a singleton which acts as a global variable that do not change with the execution of the Blended Workflow. It has fields with all specific elements that represent the access control rules. Common to all security patterns are the `resources` and `permissions` fields. Some security patterns add additional fields with their specification. The `resources` contains all existing resources of the Blended Workflow specification. This resources representation may vary depending on the pattern. The `permissions` is a function that maps `Right` x `Subject` x `resources` which represents the permissions. The state used in this specification has a new field called `log` which records every state transition previously made in a ordered way. For that purpose there is the signature `Transition`. This is useful in order to verify that the operations occurred while respecting the permissions and the invariants.

Listing 5.1: Blended Workflow Secure Specification

```
1  abstract sig Subject{}
2
3  abstract sig Rights{}
4
5  abstract sig User{}
6
7  one sig Def, Read extends Rights{}
8
9  abstract sig Transition{}
10
11 one sig AccessControlRules {
12     resources: set {Obj + FName},
13     permissions: Rights ->  Subject ->  resources,
14 }
15
16 abstract sig AbstractSecureState extends AbstractState {
17     //Log of the operations made in each transition
18     log: seq Transition
19 }
```

5.2.1.2 Secure Data Conditions

The secure data model has three possible transitions. The `defObjTransition`, `defAttTransition` and `linkObjTransition` which stores the information related to that operation. Whenever an entity definition occurs it is stored the `Obj` that represents the defined entity and the user that defined that `Obj`. In the case of an attribute definition it is stored the `FName` that represents the attribute defined, the `Obj` that

represents the attribute entity and the user. When an association between entities occurs it is stored the `Obj` entities of the association and the `FName` that represents that association. There is no need to store the `FName` that represents the inverse association because it exists as a field of the `FName`.

Listing 5.2: Secure Data Model Transitions

```
1  sig defObjTransition extends Transition{
2      dO_obj: Obj,
3      dO_usr: User
4  }
5
6  pred addObjToLog (s, s': AbstractSecureState, o: Obj, usr: User){
7      some d: defObjTransition| d.dO_obj = o and d.dO_usr = usr and s'.log = s.log.add[d]
8  }
9
10 sig defAttTransition extends Transition{
11     dA_att: FName,
12     dA_obj: Obj,
13     dA_usr: User
14 }
15
16 pred addAttToLog(s, s': AbstractSecureState, o: Obj, att: FName, usr:User) {
17     some d: defAttTransition| d.dA_obj = o and d.dA_att = att and d.dA_usr = usr and s'.log = s.log.add[d]
18 }
19
20 sig linkObjTransition extends Transition{
21     lO_objSource: Obj,
22     lO_attSource: FName,
23     lO_objTarget: Obj,
24     lO_usr: User
25 }
26
27 pred addLinkToLog(s, s': AbstractSecureState, objSource: Obj, attSource: FName, objTarget: Obj, usr:User){
28     some l: linkObjTransition| l.lO_objSource = objSource and l.lO_attSource = attSource
29                                and l.lO_objTarget = objTarget and l.lO_usr = usr and s'.log = s.log.add[l]
30 }
```

The operations specified in the secure data model are `secureDefObj`, `secureDefAtt` and `secureLinkObj`. Note that the predicate that ensures a subject has permission will vary from pattern to pattern. These operations use the same parameters as the operations already defined in the data model and additionally receive a user.

The `secureDefObj` which is shown in Listing 5.3 ensures that the user that is performing the operations exists in the access control rules and that he has permission to execute that operation through the predicate `hasDefObjPermissions`. Then the operation uses the `defObj` operation and add the operation information to the log through the `addObjToLog`.

Listing 5.3: Secure Object Definition

```
1 pred secureDefObj(s, s' : SecureState, o: Obj, usr:User) {
2     hasDefObjPermissions[s, o, usr]
3     defObj[s, s', o]
4     addObjToLog[s, s', o, usr]
5 }
```

The `secureDefAtt`uses the same approach as the `secureDefObj` and ensures that the user exists in the access control rules and has permissions to define the attribute. In order to verify the user permissions the `hasDefAttPermissions` ensures that the user has permission to define the attribute and permissions to read the object of the attribute. Then it uses the `defAtt` operation and adds the operation information to the log through the predicate `addAttToLog`.

Listing 5.4: Blended Workflow Secure Attribute Definition

```
1 pred secureDefAtt(s, s': SecureState, o: Obj, att: FName, usr:User) {
2     hasDefAttPermissions[s, o, att, usr]
3     defAtt[s, s', o, att]
4     addAttToLog[s, s', o, att, usr]
5 }
```

The `secureLinkObj` approach is similar to the previous operations. It is ensured that the user that is trying to perform the operation exists and has permissions to link the objects having permission to define the `FName` that represents the association and to read both association objects. Then the `linkObj` operation is used and the operation information is added to the log through the predicate `addLinkToLog`.

Listing 5.5: Blended Workflow Secure Object Association

```
1 pred secureLinkObj(s, s': SecureState, objSource: Obj, attSource: FName, objTarget: Obj,
2                    attTarget: FName, usr:User) {
3     hasLinkObjPermissions[s, objSource, attTarget, objTarget, usr]
4     linkObj[s, s', objSource, attSource, objTarget, attTarget]
5     addLinkToLog[s, s', objSource, attTarget, objTarget, usr]
6 }
```

The generic secure invariants are also defined in the secure data model and are grouped in a predicate called `secureDMInv`. These invariants are `ACObjDefInv`, `ACAttDefInv` and `ACLinkDefInv`, each one of these predicates ensure that respectively for every transition in the log, the users had the permissions to perform it. Note that, the verification is done for the current state and not for the state where the operation was executed because the access control does not change whenever a state transition occur. Like in the basic operations the permissions verification in these invariants change from pattern to pattern

Listing 5.6: Blended Workflow Secure Invariants

```
pred ACObjDefInv(s: AbstractSecureState){
    all do: Int.(s.log) <: defObjTransition | hasDefObjPermission[s, do.dO_obj, do.dO_usr]
}

pred ACAttDefInv(s: AbstractSecureState){
    all da: Int.(s.log) <: defAttTransition | hasDefAttPermission[s, da.dA_obj, da.dA_att, da.dA_usr]
}

pred ACLinkDefInv(s:AbstractSecureState){
    all l: Int.(s.log) <: linkObjTransition |
                            hasLinkObjPermission[s, l.lO_objSource, l.lO_attSource,
                                                    l.lO_objTarget, l.lO_usr]
}

pred secureDMInv(s: AbstractSecureState){
    ACObjDefInv[s]
    ACAttDefInv[s]
    ACLinkDefInv[s]
}
```

5.2.1.3 Secure Activity Conditions

The secure activity model uses the same state as the other models, but adds to the log the information about the execution of activities. That activity log contains all the resources existing in the pre-condition and post-condition.

Listing 5.7: Secure Activity Model Transition

```
sig activityTransition extends Transition{
    act_usr: User,
    act_PreDefObj: set Obj,
    act_PreDefAtt: set Obj -> FName,
    act_PostDefObj: set Obj,
    act_PostDefAtt: set Obj -> FName,
    act_PostLinkObj: set Obj -> FName -> Obj
}
```

As it happens in the secure data model, all the information is stored in the `log` whenever a transition between states occur, but in the secure activity model only one type of `Transition` is used. This information is used to make a subsequent verification that the user who executed the activity had permissions to do it. Whenever an activity occurs the predicate `addActivityToLog` stores in the `log` the user that executed the activity, the pre-condition entities and attributes, and all the post-condition resources that were defined through the execution of this activity.

Listing 5.8: Activity Information Addition to Log

```
pred addActivityToLog (s, s': SecureState, pre_entDefs: set Obj, pre_attDefs: set Obj -> FName,
                       post_entDefs: set Obj, post_attDefs: set Obj -> FName,
                       post_muls: set Obj -> FName -> Obj, usr: User) {
    some a: activityTransition|
        a.act_PreDefObj = pre_entDefs and
        a.act_PreDefAtt = pre_attDefs and
        a.act_PostDefObj = post_entDefs and
        a.act_PostDefAtt = post_attDefs and
        a.act_PostLinkObj = post_muls and
        a.act_usr = usr and
        s'.log = s.log.add[a]
}
```

Unlike the activity model the secure activity model groups both pre-conditions and post-conditions in one model. These ensures that all permissions verifications and log recording occurs properly. In the next paragraphs the structure of the secure activity model is represented. Note once again that the permissions verification will change from pattern to pattern.

The pre-condition contains the resources that must already be defined and the post-condition contains the resources that are created with the execution of the activity. In order to execute an activity the user needs read right over all resources in the pre-condition and def rights over all resources in the precondition. This is ensure through the predicate shown in Listing 5.9 which use the predicates defined in the secure data model to verify the permissions.

Listing 5.9: Secure Activity Pre-Condition Permissions

```
pred hasActivityResourcesPermissions(s, s': AbstractSecureState, pre_entDefs: set Obj,
                                     pre_attDefs: set Obj -> FName, post_entDefs: set Obj,
                                     post_attDefs: set Obj -> FName, post_muls: set Obj -> FName -> Obj,
                                     usr: User){
    (pre_entDefs != none) implies{
        all o: pre_entDefs| hasReadObjPermission[s, o, usrSource]
    }
    (pre_attDefs != none -> none) implies{
        all obj: pre_attDefs.FName, att : obj.pre_attDefs| hasReadAttPermission[s, obj, att, usr]
    }
    (post_entDefs != none) implies{
        all o: post_entDefs| hasDefObjPermission[s, o, usrSource]
    }
    (post_attDefs != none -> none) implies{
        all obj: post_attDefs.FName, att : obj.post_attDefs| hasDefAttPermission[s, obj, att, usr]
    }
    (post_muls != none -> none -> none) implies{
        all objSource: post_muls.Obj.FName| all attSource: objSource.post_muls.Obj
            | all objTarget: attSource.(objSource.post_muls) |
            hasLinkObjPermission[s, objSource, objTarget, attSource, usr] }}
```

In order to ensure that the user has permissions to execute an activity, there is a predicate called `hasActivityPermission`. This permissions can be granted by different means as is described in the next chapter. In this work simpler patterns of the access control the user needs to have permissions, with a subject that represents that user, over all resources of the activity. The `hasActivityPermission` uses the predicate described in the Listing 5.9 to verify that the subject has the permissions. Additionally the `secureActivity` uses the `addActivityToLog` in order to store all resources and the user that executed the activity.

Listing 5.10: Blended Workflow Secure Activity Model

```
1  pred secureActivity(s, s': AbstractSecureState, pre_entDefs: set Obj, pre_attDefs: set Obj -> FName,
2                      post_entDefs: set Obj, post_attDefs: set Obj -> FName,
3                      post_muls: set Obj -> FName -> Obj, usr: User ){
4
5      hasActivityPermissions[s, s', pre_entDefs, pre_attDefs,
6                             post_entDefs, post_attDefs, post_muls, usr]
7      preCondition[s, pre_entDefs, pre_attDefs]
8      postCondition[s, s', post_entDefs, post_attDefs, post_muls]
9      addActivityToLog[s, s', pre_entDefs, pre_attDefs,
10                     post_entDefs, post_attDefs, post_muls, usr]
11 }
```

The generic secure invariants are not the same as the ones defined in the secure data model. In the secure activity model there is only one invariant called `ACActInv` which ensures that for every log entry the user has permission to execute that activity. Like in the secure data model the verification is made for the current state and not for the state where the activity occurred because the access control rules do not change with the execution of the activity model.

Listing 5.11: Secure Activity Model Generic Invariants

```
1  pred ACActInv(s: AbstractSecureState){
2      all a: Int.(s.log) <: activityTransition |
3          hasActivityPermissions[s, s.next, a.act_PreDefObj, a.act_PreDefAtt, a.act_PostDefObj,
4                                 a.act_PostDefAtt, a.act_PostLinkObj, a.act_usr]
5  }
```

5.2.2 Specific

In the specific part exists a secure Blended Workflow specification which adds the base access control to the already existing Blended Workflow specification.

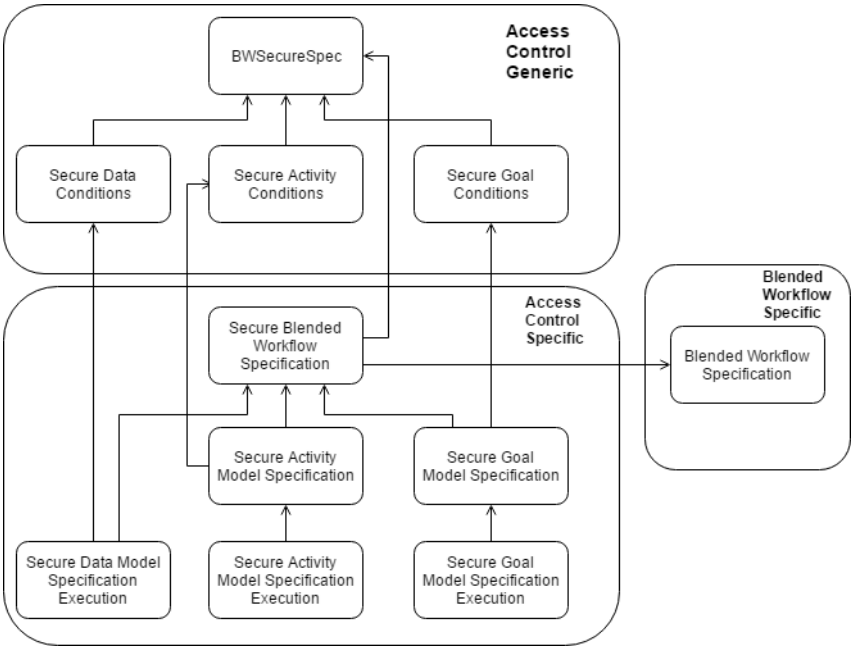

Figure 5.2: Access Control Specific Alloy Structure

5.2.2.1 Secure Blended Workflow Specification

The secure Blended Workflow specification uses the already defined specification adding only the access control specifics such as the subjects and the resources. Depending on the security pattern the subject and resources may be represented by different signatures. In some patterns domain objects are added and this requires adding new invariants and completion representations. The access control rules are also specified in this section. It is possible that the specification use on or multiple patterns, which it is the case of the majority of the examples created to ensure proper behavior of the defined patterns. The utimate goal is to create a specification in which all patterns can be used together.

5.2.2.2 Secure Data Model Specification and Execution

Like the specification of the data model without access control this one does not need to specify the operations of each different Blended Workflow specification. The representation of the execution has a signature that extends `AbstracSecureState` which is only used in this execution. First this initial secure state is defined where there is no `objects`, `fields` or `log`. There is a fact that ensures that the

transition between states only occur when of the defined operations occur. It is also specified an execution where the model achieves the complete state defined in the Blended Workflow specification and verifications which ensures that every operation respect both invariants and secure invariants. Although the data model specification and execution is similar across all patterns the operations: `secureDefObj`, `secureDefAtt` and `secureLinkObj` have different ways to ensure that a user can execute the operation and the `secureDMInv` must verify the correct execution of the operation according to each pattern verification. Additionally, in patterns that add domain objects, the invariants and complete state have different verifications from patterns that do not add domain objects.

Listing 5.12: Secure Data Model Execution

```
1  sig SecureState extends AbstractSecureState{}
2
3  pred secureInit (s: SecureState) {
4      //objects
5      no s.objects
6      //fields
7      no s.fields
8      //log
9      no s.log
10 }
11
12 fact traces {
13     first.secureInit
14     all s: SecureState - last | let s' = s.next |
15     some p: Patient, e: Episode, u: User|
16         secureDefObj [s, s', p, u] or
17         secureDefAtt [s, s', p, patient_name, u]  or
18         secureDefAtt [s, s', p, patient_address, u] or
19         secureDefObj [s, s', e, u] or
20         secureDefAtt [s, s', e, episode_reserve_date, u] or
21         secureLinkObj [s, s', p,  episode_patient, e, patient_episode, u] or
22         secureLinkObj [s, s', e, patient_episode, p, episode_patient, u]
23 }
24
25 run complete for 4 but 8 SecureState, 5 Int
26
27 assert ExecPerservesInv{
28     all s, s': SecureState, u: User|
29         secureDMInv [s] and Invariants [s]
30             => secureDMInv [s'] and Invariants [s]
31 }
32 check ExecPerservesInv for 4 but 8 SecureState, 5 Int
```

5.2.2.3 Secure Activity Specification and Execution

A – Specification The secure activity model specification is very similar to the activity model one, but instead of using the predicates for each condition it is used a predicate that groups both conditions and all verification related to the activity. The example used in the secure activity model specification is the same of the activity model specification described in Section 4.2.3.1.

Listing 5.13: Secure Activity Model Specification example

```
1 pred secureRegisterPatient(s, s': AbstractSecureState, p: Patient, usr: User) {
2     secureActivity[s, s', none, none -> none,
3                             p, p -> patient_name, none -> none -> none, usr]
4 }
```

B – Execution The Blended Workflow specific secure activity model is where the `SecureState` signature is defined. Like in the secure data model execution, there is a fact that initiates the first state without `objects`, `fields` or `log`. Besides the representation of the execution of the activities defined in the previous section, there is an assertion that ensures that for every state and any activity that occurred the invariants and secure invariants are always respected.

6

Access Control Patterns Implementation

Contents

In this chapter the security patterns and their specifications are described. Each section of the chapter represents a different pattern which follows the same approach of the base access control and is divided in generic and specific parts. These patterns are independent of each other but every new pattern is build on top of the previous ones.

6.1 User Based Access Control

The user base access control that is used give permissions directly to the user. This access control is the most simple and granular access control in this work. When defining a permission with this access control the `Subject` can be represented as a `User` signature.

In order to define permissions using users it is necessary to specify the `User` signature as an extension of `Subject`. The `Rights` of the permission can either be `Def` or `Read` and the resources can either be `Obj` or `FName` signatures. Additionally the specification of the `AccessControlRules` has a field where the users of the Blended Workflow are represented.

Listing 6.1: Security Pattern 1 Signatures

```
1 abstract sig User extends Subject{
2 }
3 one sig AccessControlRules {
4     users: set User,
5     resources: set {Obj + FName},
6     permissions: Rights -> Subject -> resources,
7 }
```

The verification of the user based access control is done by a predicate called `userBasedPermission`. This predicate has the current state, two resources, which can be either an `Obj` or an `FName`, the right and a user as arguments. When defining or reading a resource such as an `Obj` or an `FName`, the predicate must receive the user that is defining or reading the resource and the respective right. The first resource must be the resource that is being defined or read. The second resource argument should always be empty and represented as `none` with the exception of the association between two `Obj`. In that case the second argument must be the inverse of the `FName` that represents one side of the association. The predicate ensures that an permission exist where the `Subject` is a `User` and the resource over which the operation is being performed is the resource of that permission and the `user` performing the operation is the `Subject` of the permission. In the Listing 6.2 the predicate and calls executed for each verification are represented.

Listing 6.2: User Based Permissions Verification

```
pred userBasedPermission(s: AbstractSecureState, res1, res2: (Obj + FName), right: Rights, usr: User){
    one uSub: (right.(AccessControlRules.permissions).res1) <: User | usr = uSub
    (!no res2) implies {
        one uSub: (right.(AccessControlRules.permissions).res2) <: User | usr = uSub
    }
}
pred hasDefObjPermission(s: AbstractSecureState, o: Obj, usr: User){
    userBasedPermission[s, o, none, Def, usr]}

pred hasDefAttPermission(s: AbstractSecureState, o: Obj, att: FName, usr: User){
    userBasedPermission[s, att, none, Def, usr]}

pred hasLinkObjPermission(s: AbstractSecureState, objSource, objTarget: Obj, attSource: FName, usr:User){
    userBasedPermission[s, attSource, attSource.inverse, Def, usr]}

pred hasReadObjPermission (s: AbstractSecureState, o: Obj, usr: User){
    userBasedPermission[s, o, none, Read, usr]}

pred hasReadAttPermission (s: AbstractSecureState, o: Obj, att: FName, usr: User){
    userBasedPermission[s, att, none, Read, usr]}
```

6.2 Dynamic Access Control

The dynamic access control also gives permissions directly to a user, but it depends on the association of his domain representation. The user is explicitly represented in the domain through an object. Although these type of objects are not entities, they are also extensions of the signature `Obj`. A single user can have several different representations in the domain, for instance, a user can be both a doctor and a patient. On the other hand, different domain paths may bring to the same user representation, for instance, the doctor may be reached from the Episode or from the Data, where the first situation corresponds to a permission associated with the Episode resource or one of its attributes, whereas the second one corresponds to a similar permission for Data or one of its attributes. This pattern overcomes the limitation of the base access control where there was no way to define permissions based on the domain state.

In order to store their domain representations the `User` signature now has a field represented as a set of `Obj` which contains all his domain representations. Another concept added to support this security pattern is the `DomainSubject` signature. This signature has a field that represents the path, which is a sequence of `FName` between the resource where the operation is being performed and the representation of the `User` in the domain. The `DomainSubject` signature extends `Subject` and it is used in order to define permissions depending on the domain association of the `User`. Like in the previous security pattern, the target resources of this pattern can be either `Obj` or `FName` and the rights can be

either Def or Read.

Listing 6.3: User Representation with Object Field

```
abstract sig User extends Subject{
    usr_obj: set Obj
}

abstract sig DomainSubject extends Subject{
    path: seq FName
}
```

The verification of the permissions represented with the DomainSubject signature are made by the predicate dynamicResourcePermission. This predicate arguments are the current state, two Obj, an FName, the right and the user performing the operation. When the operation is the definition or reading of an Obj the second Obj and the FName in the arguments are empty and represented as none, if the operation is on an FName, the predicate does not receive a second Obj and in the case where the operation is the definition of an association between two Obj, the predicate receives the Obj signatures of both ends of the association. The verification ensures that a permission exists where the Subject is a DomainSubject and the resource is the object that is being defined and, in order to fulfill this rule, one of the user's representations in the domain must be referred by the path associated with the DomainSubject, applying this path from the resource instance for which the permission applies. Since the object to be defined does not exist in the model, the permission should be verified in the resulting state, in order to be possible to evaluate the path. The permissions for the definition of attributes follow a similar approach but the path is applied from the objects the attribute is going to be defined in. The same happens in the definition of an association but there must exist a path associated to a DomainSubject must refer the user domain representation when applied from either end of the association. The predicates for verify if a user can read an object or an attribute it is similar to the definition object but the permission must have Read right instead of Def. When performing an operation the user must have either the permissions defined in this pattern or the ones defined in the base access control.

Listing 6.4: Dynamic Permissions Verification

```
pred dynamicResourcePermission (s: AbstractSecureState, obj1, obj2: Obj , att1: FName, right: Rights,
                                usr: User){
    (!no obj2) implies{
        some dSub: (right.(AccessControlRules.permissions).(att1.inverse)) <: DomainSubject
            | some obj: usr.usr_obj | obj in reach[s, obj2, dSub.path]
        or (some dSub: (right.(AccessControlRules.permissions).att1) <: DomainSubject
            | some obj: usr.usr_obj| obj in reach[s, obj1, dSub.path] )
    }
    else (!no att1) implies{ some dSub: (right.(AccessControlRules.permissions).att1) <: DomainSubject
```

```
10              |  some obj: usr.usr_obj | obj in reach[s, obj1, dSub.path]
11      }
12      else{ some dSub: (right.(AccessControlRules.permissions).obj1) <: DomainSubject
13              |  some obj: usr.usr_obj | obj in reach[s, obj1, dSub.path]
14      }
15  }
16
17  pred hasDefObjPermission(s: AbstractSecureState, o: Obj, usr: User){
18      dynamicResourcePermission[s, o, none, none, Def, usr]}
19
20  pred hasDefAttPermission(s: AbstractSecureState, o: Obj, att: FName, usr: User){
21      dynamicResourcePermission[s, o, none, att, Def, usr]}
22
23  pred hasLinkObjPermission(s: AbstractSecureState, objSource, objTarget: Obj, attSource: FName, usr:User){
24      dynamicResourcePermission[s, objSource, objTarget, attSource, Def, usr]}
25
26  pred hasReadObjPermission (s: AbstractSecureState, o: Obj, usr: User){
27      dynamicResourcePermission[s, o, none, none, Read, usr]}
28
29  pred hasReadAttPermission (s: AbstractSecureState, o: Obj, att: FName, usr: User){
30      dynamicResourcePermission[s, o, none, att, Read, usr]}
```

6.3 Role Based Access Control

The role base access control grants the permission to roles instead of directly to users. These roles create a level of abstraction that allows the definition of permissions before the existence of users in the domain. The role main purpose is to group permission common to many users, knowing that, there is representation of the roles each user has. In order to define permissions the `Subject` can now be represented as a role. This permission grants the right over a resource to all users that have the role in that permission.

In this security pattern it is added the concept of role. This concept is represented by a signature called `RoleSubject`.The `AccessControlRules` signature now have a field represented by a set of `RoleSubject` which contains all the roles in the domain and another field which is a function that maps users to their respective roles.

Listing 6.5: Role Specification

```
1  abstract sig RoleSubject extends Subject{}
2
3  one sig AccessControlRules {
4      users: set User,
5      roles: set RoleSubject,
6      u_roles: users -> set roles,
7      resources: set {Obj + FName},
```

```
8       permissions: Rights -> Subject -> resources,
9   }
```

The verification of the role based access control is made by the predicate `roleBasedPermission`. This predicate is similar to the one used in user based access control. It has the current state, two resources, which can be either an `Obj` or an `FName`, the right and a user as arguments. When defining or reading a resource such as an `Obj` or an `FName`, the predicate must receive the user that is defining or reading the resource and the respective right.In order to allow a user to perform an operation there must be a permission where the `Subject` is a `RoleSubject` and the target the resource over which the operation is being made. Additionally the user that is performing the operation must have the role existing on the permission. The verification is the same except for the definition of an association where there must be a permission for each `FName` signatures that represent the association.

Listing 6.6: Role Based Permission Verification

```
1  pred roleBasedPermission(s: AbstractSecureState, res1, res2: (Obj + FName), right: Rights, usr: User){
2      some role: (right.(AccessControlRules.permissions).res1 <: RoleSubject)
3          | role in usr.(AccessControlRules.u_roles)
4      (!no res2) implies {
5          some role: (right.(AccessControlRules.permissions).res2 <: RoleSubject)
6              | role in usr.(AccessControlRules.u_roles)
7      }
8  }
```

6.4 Access Control Over Model Operation

The three patterns represented until this point all use basic resources as the target of their permissions. The access control over model operation change that. Instead of permission over basic resources, they are over operations. These operations are the goals and activities existing in the Blended Workflow. For example, when registering a patient, the hospital worker, instead of having a permission to define the Patient, another to define his name and another to define his address he has only one that allow him to perform the activity register patient. This pattern can be used in conjunction with the three previous security patterns. The permissions can have as resource an operation and the `Subject` represented as a `User`, a `DomainSubject` or a `RoleSubject`. The operations are now identified by their name.

This security pattern adds the concept of operation. This concept is represented by the signature `Operation`. With the permission now being defined over operations, the `AccessControlRules` field where the resources are represented now contains the `Operations` signatures in addition to the `Obj` and `FName`. The addition of the signature `Operation` causes a modification in the predicates `secureActivity` and `secureGoal` which in addition to all the arguments received previously, now receives an `Operation`

signature which is the signature identifier. The arguments added to these two predicates have no impact in the previous security patterns which can use these new predicates with the operation argument empty. This pattern causes another modification in the `Transition` extensions for both activity and goal which now have an extra field that stores the operation. The predicates `addActivityToLog` and `addGoalToLog` starts to add the operation to the log. This is used for the assertions that demonstrate the correct execution of the operations.

Listing 6.7: Operations specification

```
1 abstract sig Operation{}
2
3 pred secureActivity(s, s': AbstractSecureState, pre_entDefs: set Obj, pre_attDefs: set Obj -> FName,
4                      post_entDefs: set Obj, post_attDefs: set Obj -> FName,
5                      post_muls: set Obj -> FName -> Obj, op: Operation,  usr: User )
```

The permission verification in this pattern is done by the predicate `hasExecOperationPermission`. This predicate uses the already defined predicates for the user based access control and the role based access control in order to verify the cases where the subject is either a `User` or a `RoleSubject` with the difference that the arguments that represent resources in those predicates can now be `Operation`. When verifying the permission of user based or role based there must exist a rule where the subject is respectively a `User` or a `RoleSubject` and the resource is the operation being performed. Like in their respective security patterns if the permission subject is a `User` the user performing the operation must be the one in the permission, if the permission subject is a `RoleSubject` the user performing the operation must have the role in the permission. For the permissions where the subject is a `DomainAssociation` a new predicate is introduced. This predicate is called `dynamicOperationPermission` and besides the current state and the user performing the operation it also receives the `Operation` that is being performed and a set of `Obj`. This set contains all objects and attributes objects of the conditions of the operation. In order to execute an operation there must exist a permission where the subject is a `DomainSubject` which path leads from at least one object of the set of objects of the operation to the user representation in the domain.

Listing 6.8: Access Control Over Model Operation Permission Verification

```
1 pred hasExecOperationPermission (s: AbstractSecureState, objs: set Obj, op: Operation, usr: User){
2     userBasedPermission[s, op, none, Def, usr] or
3     dynamicOperationPermission[s, op, objs, Def, usr] or
4     roleBasedPermission[s, op, none, Def, usr]
5 }
6
7 pred dynamicOperationPermission(s: AbstractSecureState, op: Operation, objs: set Obj, right: Rights,
8                                  usr: User){
```

```
9       some o: objs| some dSub: (right.(AccessControlRules.permissions).op) <: DomainSubject
10          | some uObj: usr.usr_obj| uObj in reach[s, o, dSub.path]
11  }
```

6.5 Privilege Propagation

The privilege propagation allows a user to share the permissions he has to another user under certain circumstances. The propagation depends on a rule which is represented by a function that maps `Operation` to `RoleSubject`. The idea is that only when performing certain operations can occur the propagation and only for users that have certain role. When propagating privileges it is not required that the source user has permissions over all resources of the operation. There may be cases where the source user has only part the permissions necessary to perform an operation and the target user of the propagation has the rest.

This pattern do not add new signatures to the access control but causes changes in already existing predicates. First of all it is added a `User` argument to the `secureActivity` and `secureGoal` predicates. This new user is the user to whom the privileges are propagated. The signature `ActivityTransition` now has an additional user and so the `addActivityToLog`. The same happens for the goal model.

The permission verification in an operation is made with the already existing predicates. In order to perform an activity with privilege propagation either the `hasExecOperationPermission` predicate is satisfied for the source user or the target user. It is not mandatory that the permissions to perform the operation target be the `Operation`. It is possible that the source user has permission over each resource, the second has the permissions or the combined permissions of the two are allow to perform the operation. This is achieve through use `hasActivityResourcesPermissions` which now receives two users as arguments. The modification to the predicate ensures that for each resource either the source user has permissions or the target user has permissions. This permission verification to the basic resources can be satisfied with any of the first three security patterns.

Listing 6.9: Secure Activity Specification

```
1  pred secureActivity(s, s': AbstractSecureState, pre_entDefs: set Obj, pre_attDefs: set Obj -> FName,
2                      post_entDefs: set Obj, post_attDefs: set Obj -> FName,
3                      post_muls: set Obj -> FName -> Obj, op: Operation,  usrSource, usrTarget: User  )
```

7

Evaluation

Contents

In this chapter an example is described in order to show the proper functioning of the access control specification achieved in this work. In this example I use as example a medical appointment case and it contains all the security patterns with the exception of the user based, which is similar to the role based but without the indirection created by the role. The execution is demonstrated with the activity model of the Blended Workflow.

This chapter is divided in three parts. The first part contains the description of the example and the way the resources are grouped into activities. The second has the specification of the example model in Alloy. This part also contains the specification of all permissions rules which allows a proper execution of the activity model. The third part contains the execution and its results, demonstrating the functioning of the access control.

7.1 Example Description

The example used to demonstrate the access control correct specification is a basic medical appointment workflow. There is an `Episode` which represents an appointment in the hospital. This `Episode` is associated to a `Patient` in a way that each `Episode` has one `Patient` and a `Patient` can have zero or many `Episode` instances. A `Patient` has the attributes `patient_name` and `patient_address` and an association of one to one with an `Patient_User` entity. This `Patient_User` is used to represent a user in the domain. On other hand an `Episode` has three attributes. One for the appointment date represented by `episode_reserve_date`, other for the check in represented by `episode_checkin` and the last one being for the checkout represented by `episode_checkout`. The Episode has `Data` referent to the `Patient` health condition and a `Report` which is done after the medical examination. The `Episode` can only have one `Data` and a `Data` can only correspond to an `Episode`. The same happens with the association between the `Episode` and the `Report`. The `Data` has four attributes which correspond to the patients height, weight, blood pressure and physical condition represented respectively by `data_height`, `data_weight`, `data_blood_pressure` and `data_physical_condition`. The `Report` only has one attribute that corresponds to the description of the `Episode`. This is represented by the attribute `report_description`. The checkout can only be done after the check in and the elaboration of the report description, and the report description can only by done after the blood pressure measurements. These three constraints are the dependences of the example. Additionally every `Episode` has at least one `Doctor_User`, but a `Doctor_User` can have zero or more `Episode` instances.

The example resources are grouped in six activities. These activities are: Register Patient, Book Appointment, Check In, Checkout, Collect Data and Write Report. The Register Patient activity consists on the definition of an instance of the entity `Patient`, its attributes and the association between the entity `Patient` and the `Patient_User` which represents a user. This activity execution has no restrictions or

constraints attached.

The Book Appointment consists on the definition of an `Episode` instance, the `episode_reserve_date` attribute and the association between the `Episode` and the respective `Patient` and `Doctor`. `Episode` instances can only be created for already existing `Patient` instances, this forces the occurrence of a Register Patient activity before the Book Appointment activity occurs. The same happens to the association with `Doctor_User`, but by default all `Doctor_User` representations on the domain already exist in the domain when the execution start.

The Check In activity consists on the definition of the attribute `episode_checkin`. This activity can only be done if an `Episode` instance is already defined, meaning that it is necessary that the Book Appointment occurs before the Check In activity.

The Collect Data activity consists on the definition of an instance of `Data` and its attributes `data_height`, `data_weight`, `data_blood_pressure` and `data_physical_condition`. Additionally the association between the `Data` and its respective `Episode` is also defined in this activity. This means that, in order to have an instance of `Episode` defined, the Book Appointment activity must be executed before the Collect Data.

The Write Report activity consists on the definition of an instance of `Report`, the `report_description` attribute and the association between the `Report` and its respective `Episode`. The `report_description` attribute depends on the `data_blood_pressure` attribute. This dependence and the need to have an `Episode` already defined means that the Write Report activity can only occur if both Book Appointment and Collect Data occurred before.

The Checkout activity consists on the definition of the attribute `episode_checkout`. This attribute depends on the attributes `episode_checkin` and `report_description`. These dependences and the need to have an `Episode` already defined causes the Checkout activity to occur after the activities Book Appointment, Check In and Write Report.

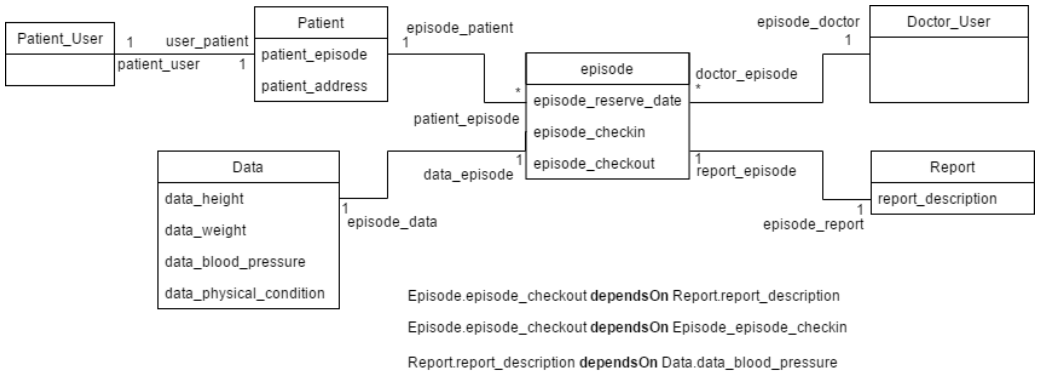

Figure 7.1: Access Control Demonstration Example

In order to execute the activity model respecting the access control, it is necessary to define the users,

roles and permissions for each resource. In this example there are four users: Alice, Bob, Carl and Dan. Each of these users has a different role. Alice has the role Doctor, Bob as the role Nurse, Carl has the role Receptionist and Dan the role Patient. The Patient role has permissions to read the Patient entity and its attributes patient_name and patient_address. Additionally the role Receptionist has definition permission over the activity Register Patient which allows the users with that role to execute that activity

The Receptionist also has permission to define Episode, the episode_reserve_date attribute and the associations between Episode and Patient and Episode and Doctor_User. In order to perform the Book Appointment activity the Receptionist role also needs to read both Patient and Doctor_User.

The Collect Data activity can be done by a Nurse. Additionally a Nurse can propagate its privileges to a Doctor . The Nurse role need to have permission to define the Data entity, the attributes data_height, data_weight, data_blood_pressure and data_physical_condition, and for the association between Data and Episode. Additionally the Nurse role also need permission to read the Episode entity.

The Write Report activity can only be done by users with the role Doctor and with a domain association to the Episode instance associated with the Report. Unlike most of the other permissions the permission to execute the Write Report activity has a DomainSubject instead of a Role. This DomainSubject contains a path form the resources that is being defined to the user representation in the domain which in this case is the path from Report to Doctor. Note that the permission maps an operation instead of the resources which grant the user to define the resources only in the context of that activity.

Both Check In and Checkout activities can be done by the Receptionist or the Patient but the patient can only execute those activities if he is the Patient of that Episode. So the role Receptionist have permission to define the attributes episode_checkin and episode_checkout and the Patient must have a domain association to the Episode where those attributes are being defined.

7.2 Alloy Representation

The Alloy representation of the example has several components. These components are the resources, the dependencies, the complete state, the structure invariants, the activities specification and the access control rules. The access control rules specification contains not only the permission rules but also the subjects such as roles, users and users domain representations.

The resource representation is made through the use of signatures. All entity types has a signature that extends Obj. In our example there are six entity types: Patient, Patient_User, Episode, Doctor_User, Data and Report. Each entity attribute is specified an a singleton signature that extends FName. Although the attributes are a singleton it is possible to differentiate different instances by the entity instance to which they are associated. The associations between entities are also defined in the

same way as the attributes but with some additional information. This association have three fields that need to be specified. These field are the `minMul` which represents the minimum multiplicity in this side of the association, the `maxMul` which represents the maximum multiplicity and `inverse` which contains the `FName` which represents the other end of the association. The example specification also contains dependencies between attributes. The `episode_report` depends on the `data_blood_pressure` and the `episode_checkout` depends on both `report_description` and `episode_checkin`. The representation of these dependencies is achieved through the use of signatures which extend `Dependency`. These representations of dependency have four fields, which are: `sourceObj`, `sourceAtt`, `sequence` and `targetAtt`, that must also be specified. In the Listing 7.1 is the representation for the entity `Report` and all its attributes, associations and dependencies.

Listing 7.1: Report Alloy Representation

```
1  sig Report extends Obj {}
2  one sig report_description extends FName {}
3  one sig report_episode extends FName {}
4
5  fact relations {
6      report_episode.minMul = 1
7      report_episode.maxMul = 1
8      report_episode.inverse = episode_report
9  }
10
11 sig report_description_data_blood_pressure_dependence extends Dependence {}
12
13 fact dependencies {
14     report_description_data_blood_pressure_dependence.sourceObj = Report
15     report_description_data_blood_pressure_dependence.sourceAtt = report_description
16     report_description_data_blood_pressure_dependence.sequence = 0 -> report_episode + 1 -> episode_data
17     report_description_data_blood_pressure_dependence.targetAtt = data_blood_pressure
18 }
```

The next step of the representation of the example in Alloy is the complete state and structural invariants. Both these components are based on the resources definition made previously. The structural invariants ensure that for the resources previously defined four invariants are respected from the beginning to the end of the execution. First there should be no fields in each entity except the specified. In the case of the `Report` there should be no fields except the `report_description` and `report_episode`. The second invariant ensures that the maximum multiplicity is not exceeded in associations between entities. The third ensures the bidirectionality of an association. The fourth ensures that all dependences are preserved during the execution, meaning that an attribute that has a dependence on other can only be defined after the attribute it depends on is defined.. Note that although the definition of entities, attributes and association is made through signatures, the connection between resources is ensured with the invariants. On

other hand there is the complete state, which is an example state that the execution must achieve. In this example the complete state should have one instance of each entity defined and their attributes and associations as well. Additionally, in this state the minimum and maximum multiplicity in associations, the bidirectionality and the dependences must be respected. In the Listing 7.2 is the representation of the `Report` invariants and complete state. Note that those representations contain every resources and the Listing 7.2 is only part of it.

Listing 7.2: Invariants and Complete State Alloy Representation

```
1  pred complete {
2      one s: AbstractState |
3          #Report <: s.objects = 1 and
4          attributesDefined [s, Report, report_description] and
5          multiplicityRule [s, Report, report_episode] and
6          bidirectionalRule [s, Report, episode_report, Episode, report_episode] and
7          checkDependence [s, Report, report_description_data_blood_pressure_dependence]
8  }
9
10 pred Invariants(s: AbstractState) {
11     noExtraFields [s, Report, report_description + report_episode]
12     noMultiplicityExceed [s, Report, report_episode]
13     bidirectionalPreservation [s, Report, episode_report, Episode, report_episode]
14     checkDependence [s, Report, report_description_data_blood_pressure_dependence]
15 }
```

The access control rules define the permissions. As said before a permission is a triple right X Subject X resource. The right is either `Def` or `Read` for definition or reading respectively. The subject can either be `User`, `RoleSubject` or `DomainSubject`. The users of this example are `Alice`, `Bob`, `Carl` and `Dan`. These user are represented as singleton signatures which extend `User`. The roles used in this example are `Doctor`, `Nurse`, `Receptionist` and `Patient`. The roles are also represented as singleton signatures but extend `RoleSubject`. The `DomainSubject` instances existing in this example are the `ReportDoctor` and the `EpisodePatient`. The signature `ReportDoctor` has a field with the path from `Report` to `Doctor`, and the signature `EpisodePatient` has a field with a path from `Episode` to `Patient`. With all the possible `Subject` representations described, it is time to describe the possible resources. Besides the basic resources like entities or attributes the permission resources can also be operations like activities or goals. The six activities of this example are represented as singleton signatures which extend the `Operation`.

The access control rules is represented by a singleton signature called `AccessControlRules`. This signature has six fields: `users`, `roles`, `u_roles`, `resources`, `permissions` and `delegation`. The `users` fields contains all four users and the `roles` field all four roles. The `u_roles` field contains a map from the

users to their roles. In this example Alice has the role Doctor, Bob as the role Nurse, Carl has the role Receptionist and Dan the role Patient. The field resources contains all entity types, all attributes and all operations. The delegation only contains a function that maps the operation Collect Data to the role Nurse. The permissions field contains all the permissions described in the beginning of this section. The Listing 7.3 shows the representation of all Subjects and the AccessControlRules. Note that this listing only shows a portion of the AccessControlRules content.

Listing 7.3: Access Control Rules Alloy Representation

```
1  one sig Alice, Bob, Carl, Dan extends User{}
2
3  one sig R_Doctor, R_Nurse, R_Receptionist, R_Patient extends RoleSubject{}
4
5  one sig registerPatient, bookAppointment, checkin, checkout, collectData, writeReport extends Activity{}
6
7  fact acrules{
8      AccessControlRules.users = {Alice + Bob}
9      AccessControlRules.roles = {R_Doctor + R_Nurse}
10     AccessControlRules.u_roles = {Alice -> R_Doctor + Bob -> R_Nurse}
11     AccessControlRules.resources = {
12         Data + data_height + data_weight + data_blood_pressure + data_physical_condition + data_episode +
13         Report + report_description + report_episode +collectData + writeReport
14     }
15     AccessControlRules.permissions =
16         {Def -> { (DomainReportDoctor -> {writeReport}) +
17                 (R_Nurse -> {Data + data_height + data_weight + data_blood_pressure +
18                             data_physical_condition + data_episode + episode_data})}
19         +Read -> {(R_Nurse -> {Episode}) +(R_Doctor -> {Episode})}
20         }
21     AccessControlRules.permissions = {collectData -> R_Doctor}
22 }
```

As said before there are six activities and in order to represent them, the predicate called secureActivity is used. This predicate receives the activities pre-condition and post-condition resources, the operation performed and the user or users that perform the activity. In the example described the activity Write Report consists on defining the Report entity, the report_description attribute and the association between the Report and the Episode. In order to call the predicate secureActivity for the Write Report operation it is necessary to pass an entity of Episode and a function mapping Data to the data_blood_pressure as pre-conditions. This pre-conditions are what needs to be defined in the execution of the Blended Workflow in order to be possible to execute the operation. As post conditions the predicate needs all the resources that are being defined in the activity which include all resources related to the Report including both sides of the association between Report and Episode. The Listing 7.4 shows the representation of the activity Write Report.

Listing 7.4: Write Report Alloy Representation

```
pred writeReport(s, s': AbstractState, e: Episode, r: Report, u:User) {
    secureActivity[s, s', e, none->none,  r, r -> report_description, (r -> report_episode -> e) + \
}
```

7.3 Results Demonstration

The execution of the example uses a scope of seven states. These seven states are the minimum required in order to be possible to execute all six activities at least one time. The execution of these six activities leads to the achievement of the complete state in which all entities must have one instance and all its attributes and associations defined. Although the example complete state has a instance of each entity, it is possible to define a complete state with two or more instances for an entity. The change in the complete state definition leads to a change in the scope of the execution of the workflow.

Before executing the workflow in order to achieve the complete state, the initial state must be defined. In the example the initial state has only one entity in the domain. This entity is `DoctorAlice` which is a representation of the user `Alice`, which has the role `Doctor`, in the domain. There are no fields in the domain in the initial state. With the initial state defined, the execution of the activities can now occur according to the access control rules defined previously. Each execution causes a transition between states. The number of transitions until achieves the complete state plus one is equal to the scope provided for state. The six activities that cause transition can be executed in any order taking into account that all dependences and activity pre-conditions are respected.

Listing 7.5: Example Execution

```
pred init (s: SecureState) {
    s.objects = {DoctorAlice}
    no s.fields
    no s.log
}

fact traces {
    first.init
    all s: SecureState - last | let s' = s.next |
    some p: Patient_Actor, e: Episode, d: Data, r: Report, doc: Doctor_Actor, u1, u2: User|
        registerPatient[s, s', p, u1] or
        bookAppointment[s, s', p, e, doc, u1] or
        checkin[s, s', e, u1] or
        checkout[s, s', e, u1] or
        collectData[s, s', e, d, u1, u2] or
        writeReport[s, s', e, r, u1]
}

```

```
19 run complete for 5 but 7 SecureState, 5 Int
```

The correct execution of the example achieved the specified complete model. The execution generated an instance of the model where the user `Carl` executed the activity Register Patient, then `Carl` executed the activity Book Appointment, the third activity, Collect Data, was executed by `Alice` and `Bob`, after that `Carl` executed the Check In activity, then `Alice` executed the activity Write Report and the last activity, which was Checkout, was executed by `Carl`. Reviewing all activities one by one, the correct execution of the model according to the defined permissions can be observed. The first activity was Register Patient and was executed by the user `Carl` who has the role `Receptionist` which grant him permissions to execute the operation. The second activity was also executed by `Carl`. With the role receptionist he has permissions to define all resources that need to be defined in the Book Appointment activity. The third activity was executed by `Alice` with the privilege propagation by `Bob` which respectively have the roles `Doctor` and `Nurse`. The `Nurse` role grant `Bob` the permission to define all resources in the Collect Data activity and there is a rule in the access control rules which allows this specific activity to be performed by a user with the role `Doctor` when another user with privileges share them. The fourth activity was done by `Carl` which has the role that grants him permission to define the `episode_checkin` resource. Note that the Collect Data activity was executed before the Check In. That happened because in any place of the model was stated that the Collect Data activity could only be done after the Check In neither through dependences or activities pre-conditions. The fifth activity was executed by `Alice`. The permissions only allow users with a domain representation associated to the `Episode` to perform the operation Write `Report` which is the case of `Alice`. The last activity was Checkout and was executed by `Carl` which through his role has permission to execute the activity.

When observing different instances of the execution of this model, it can be observed that different users executed some activities. With the exception of Register Patient, Book Appointment and Write Report, which were always executed by the same user in different instances, it was achieved instances of the complete model where the Check In and Checkout where either executed by either `Carl`, which have the role `Receptionist` or `Dan` which has a domain representation associated to the `Episode`. Additionally the Collect Data was either executed only by `Bob` or `Alice` and `Bob`.

```
Executing "Run complete for 5 but 5 int, 7 SecureState"
  Solver=sat4j Bitwidth=5 MaxSeq=5 SkolemDepth=1 Symmetry=20
  1050320 vars. 9939 primary vars. 2359386 clauses. 28918ms.
  Instance found. Predicate is consistent. 1139ms.
```

Figure 7.2: Access Control Execution Example

7.4 Comparison With Related Work

The related work described in Chapter 3 focus on three different works. These works have in common the specification of security patterns or requirements that help the authors to achieve an access control which overcome difficulties encountered in their scope. In this section there is a comparison between the related work requirements or patterns and the access control achieved with this work.

The authors of [5] has six requirements specified. The first requirement is the role base access control. This requirement grants access to the users of a particular group that have a specific role, the group is given by a relation to the resources being accessed. The second is process instance based user group. This requirement grants access to the users of a particular group that have a specific role, the group is given by a relation to the resources being accessed. Although it is not totally similar, there is a representation of this requirement in this work but instead of using the role plus the association, only the domain association is used in this specification. In order to grant access based in both the domain association and the role would be necessary to extend the proposed access control to support more complex permissions. This permissions would compose two or more simple permissions and all of them should be respected in order to access a resource. The third requirement is the task based. This requirement in addition to the role, only allow users to access the resources when performing certain tasks. This requirement is specified through the permission definition. Having permissions over certain tasks, or my case, over activities or goals means that the user performing certain activity can only define or read the resources of that activity in the context of the activity execution. The fourth requirement is that permissions must be defined over resources. This requirement is used in the work through the definition of permission. The permission definition achieved in this work is `Rights` X `Subject` X resources. The permissions can be defined over all resources of the domain. The fifth requirement is the privilege propagation and is also specified in this work. The last requirement is the dynamic authorization. This requirement is not supported by my solution. This requirement states that permissions can change when certain situations occur in the workflow. The implementation I used for the access control rules is static and once defined it is not changed. In order to specify this requirement it would be necessary to change the access control rules to be associated to each state of the execution and there could be situations that change the access control rules when transitioning between states.

In [4] the focus is on the common security patterns in web applications. They describe seven security patterns. The first, which is ownership, is not represented in this work because does not add benefits in the context of the workflow execution. Nevertheless, if it were implement, the access control execution should be extended to create an association between the object owned by the user and the user representation in the domain whenever a resource is defined. The third pattern is public objects. In this work there weren't considered public objects. The third pattern is authentication. This pattern is also not used because it is assumed that every user is authenticated during the execution. The fourth

pattern is explicit permissions. These pattern is represented in this work in the access control rules and all permissions in this work are explicit. The fifth pattern is related to user profiles which weren't also considered in this work. The sixth pattern is the administrator. Although there is no role administrator on the example used in the demonstration of the access control achieved, it is possible to define a role or a user that has permissions over all resources and operations existing. The last security pattern is the explicit roles. All roles existing in the workflow execution are represented in the access control rules defined in the design time and carried till the end of the execution.

The work presented in [7] contains seven requirements. The first pattern is the dynamic activation of roles and permissions. Although there are no activation in the proposed access control, the role and permissions verification is executed every time a user tries to execute an activity. The three next requirements are authorization depending on data, on relations and on object state. From these three requirements only the authorization depending on relations is included in my model of access control. In the purposed access control the authorization depending on the data of the user is achieved with the domain representation of a user and its association. The authorization depending on object state is also not included due to the absence of object states. The fifth requirement is authorization depending on the resources value. This requirement is not included in the specified access control due the fact that there is no values associated to attributes, instead it is only represented if they are defined or not. The last two requirements are related to performance and real time permission and role authorization and focus on concurrent execution of activities which does not happen in the achieved access control.

7.5 Validation

In this section it is described the validation of the access control. The specification of the access control has three different parts:the user model, the rules model and the Blended Workflow. The validation is focused on each of these three parts. The user model contains the specification of all things subject related including the users representation in the domain and their association to other domain entities. The rules model contain all rules such as permission definition and rules that support the security patterns like the rules that state which operations can be propagated and to whom. The Blended Workflow specification contains all the resources existent in the domain and their possible executions. For validation purpose I considered that two instances of Blended Workflow, with the same resources and different operations specification are distinct but can be equivalent from the execution point of view, if they can generate the same complete state.

The first step is the verification of the correct execution of the activities. The example in this validation uses the user model, rule model and Blended Workflow specified previously. After these first verifications, there are model variations for which it is verified if if a complete state can be executed when one of

the user model, rule model or Blended Workflow changes. Although I am going to analyse the different situations where the model is changed, my main emphasis is on the blended workflow model because it is where my approach differs from other approaches.

Figure 7.3 represents the structure of a Blended Workflow with access control specification. The `User Model` contains all the specification related to users. The `Rule Model` contains all specification related to permissions and rule related to the security patterns. The `Blended Workflow` contains all the specification of the domains such as resources and the way they are grouped into operations. The `DomainSubject` are the representation of the user in the domain, this representation is defined in the `User Model` but applied to the domain specified in the `Blended Workflow`. The `Subject` is defined in the `User Model` but is applied in the `Rule Model` in order to define permissions. The `Subject` can have different representations such as user, role or domain representation of a user. Resources are the entities and attributes of the domain. They are defined in the `Blended Workflow` but are used to define define permissions in the `Rule Model`.

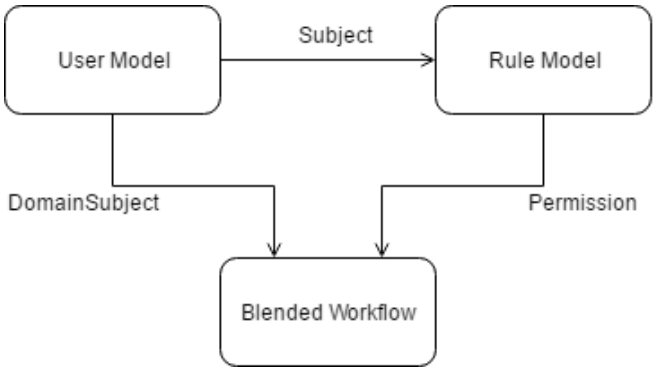

Figure 7.3: Validation Models

7.5.1 Operations Verification

In this section the validation of the operations of the Blended Workflow is described. This validation is produced in Alloy language through the use of `assert` statements as shown in the Listing 7.8 in the end of this section. This validation purpose is to prove that the operations can only be executed by the users with the required access control rules. The operations provided as example are the activities described in the previous example. The scope used for this verifications is the same as the scope used to achieve a complete state. Although some verifications could be done with a reduced scope, for example the Register Patient activity could be done with a scope of only two states, i've decided to do all verifications with the same scope.

The activities Register Patient and Book Appointment can only be executed by a user that has a `Receptionist` role. The first verification made to these operation is that whenever a one of them occurs,

the user that executes that operation must have the Receptionist role. The second verification made is that a particular user cannot execute this operation. The chosen user was Alice, which has a single role, Doctor, and does not have permissions. The assertion made states that there should be no state where Alice executes either one of these activities.

Both Check In and Checkout activities can be done either by a user with the role Receptionist or the user with a domain representation associated to Episode. The DomainSubject that represents the association between the Episode and the user representation in the domain is called DomainEpisodePatient. The first verification for both activities is that in whenever one of this activities occur, the user must either have a role Receptionist or a user representation that is in reach of the path represented by the DomainEpisodePatient applied from the Episode where the check in or checkout is performed. The second verification is whether a particular user can or not execute these activities. This states that there should be no state where a user, in this case Alice, can execute these activities.

The Collect Data Activity can be done either by a user with Nurse role or by a user with a role Doctor to which the nurse user delegates his permissions. The Collect Data has two predicates that represent its execution, one represents the execution only by one user and the other represents the execution of a user to whom the privileges where propagated. The first verification is for the case when the user executes the activity in the normal way. This verification states that whenever a Collect Data activity occurs, the user that executed this activity must have the role Nurse. The second verification is for that case where the activity was executed with privilege propagation. Since the activity can only be propagated for users with the role Doctor it is verified that the user to whom the permissions where propagated has the role Doctor. The third verification is that a user cannot execute this activity, and that there should be no state where a user, in this case Alice, executes the activity the normal way.

The Write Report activity can only be done by the a user that has a domain representation associated to the Report. The DomainSubject that represents the association between the Report and the user representation in the domain is called DomainReportDoctor. The first verification ensures that whenever a Write Report activity occurs, the user that executed that activity has a representation in the domain that can be reached by the path represented by DomainReportDoctor starting in the Report that is being defined. The second verification is similar to the ones made to other activities with the exception that Alice can execute this operation due to her domain representation. This verification states that there should be no state where this operation is executed by Dan which has no domain representation associated to the Report.

Listing 7.6: Activities Verification

```
assert onlyReceptionistCanRegisterPatient{
    all s: SecureState, p: Patient, pu: Patient_User, u:User|
            registerPatient[s, s.next, p, pu, u] => R_Receptionist in u.(AccessControlRules.u_roles)
}

assert AliceCantRegisterPatient{
    all p: Patient, pu: Patient_User|
        no s: SecureState | registerPatient[s, s.next, p, pu, Alice]
}

assert onlyReceptionistOrPatientCanCheckIn{
    all s: SecureState, e: Episode, u:User|
        checkin[s, s.next, e, u]
            => R_Receptionist in u.(AccessControlRules.u_roles)
            or some obj: u.usr_obj| obj in reach[s.next, e, DomainEpisodePatient.path]
}

assert onlyNurseCanCollectData{
    all s: SecureState, e: Episode, d: Data, u: User|
        collectData[s, s.next, e, d, u]
            =>  R_Nurse in u.(AccessControlRules.u_roles)
}

assert onlyNurseCanPropagateToDoctor {
    all s:SecureState, e:Episode, d: Data, u1, u2: User|
        collectData[s, s.next, e, d, u1, u2]
        => R_Nurse in u1.(AccessControlRules.u_roles) and R_Doctor in u2.(AccessControlRules.u_roles)
}

assert onlyDoctorCanWriteReport{
    all s: SecureState, e: Episode, r: Report, u: User|
        writeReport[s, s.next, e, r, u]
            => some obj: u.usr_obj| obj in reach[s.next, r, DomainReportDoctor.path]
}
```

The verifications made for each operation to ensure that the user that executes them has the correct permissions has the same function of the logs existent in each state. Note that the logs verification are generic and apply for any combination of Blended Workflow, user model and rule model specification. On other hand, the verification used for each operation is specific to the instances of Blended Workflow and rule model of the example used. This means that if one of these instances change, the log could still be used to verify the new operations or permissions but these assert statements needed to be changed in order to adapt to the new specification.

7.5.2 Specification Changes

In this section, verifications are made to understand if it is possible to achieve a complete state when a part of the specification: user model, rules model or Blended Workflow is changed. First it is de-

scribed the verification made with different user models, followed by the different rule models and different Blended Workflow instances, where the last one has a more exhaustive verification than the other two. The example used in the previous verification is good for verify different types of operations but for testing different user model, rule model and Blended Workflow instances a more simple one is used.

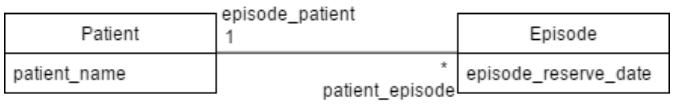

Figure 7.4: Validation Example

The example used for the next validation is a compact version of the one presented before where there are only two entities as shown in Figure 7.4. The entity `Episode` has a single attribute, which is called `episode_reserve_date` and the entity `Patient` also has a single attribute called `patient_name`. Between this two entities there is an association where a `patient` can have many `Episode` and each `Episode` has a `Patient`. There are two activities, the first called Register Patient defines an instance of the `Patient` entity and its attributes. The second called Book Appointment defines an instance of `Episode`, its attribute and the association between *Episode* and `Patient`. There is a single user named `Carl` which has a single role `Receptionist` which has permission to define each resource of this example and permission to read `Episode` and `Patient`. There is a complete predicate which contains one instance of each resources and can be achieved by the execution of this Blended Workflow, rule model and user model.

7.5.2.1 Different User Models

In this verification several instances of `User Model'` are tested to verify if it is still possible to execute the Blended Workflow and achieve a complete state and in which situations can't. In this first case the `Rule Model` and `Blended Workflow` used are described previously and do not change for any instance of the `User Model'`.

There were made verification for five different instances of `User Model'`. In the first `User model'` a new user, `Alice`, was added with a single role `Doctor`. In the second `User Model'` a role was added along side with the already existing role. In third, the user `Carl` was removed. In the fourth the role `Receptionist` was removed from `Carl` roles. In the fifth the user `Carl` was replaced by a new user, `John`, with the same roles `Carl` had. The verification made was the same for each `User Model'` and stated that for the `User Model'`, the `Rule Model` and the `Blended Workflow` represented as `execution` in the Listing 7.7 there could not be achieved the complete state. This was only verified for the case were

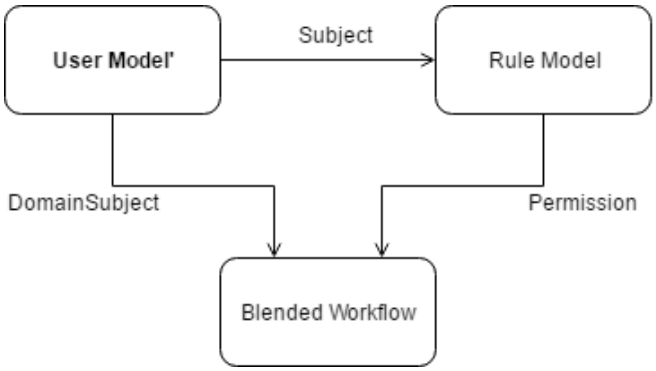

Figure 7.5: Different User Models Approach

the user `Carl` was removed or the role `Receptionist` was removed from the user. Meaning the complete model could still be achieved with different `User Model` as long as an user with the role `Receptionist` existed. This is only lightly explored as the example chosen is very basic and, given a more complex instance of `User Model` other verifications can be made.

Listing 7.7: User Model Modification Example

```
1  pred userModelNewUserAdded{
2      AccessControlRules.users = {Carl + Alice }
3      AccessControlRules.roles = {R_Receptionist + R_Doctor}
4      AccessControlRules.u_roles = {Carl -> R_Receptionist + Alice -> R_Doctor}
5  }
6
7  assert differentUserModelCantAchieveComplete1{
8      userModelNewUserAdded and ruleModel and execution
9          implies !complete
10 }
```

7.5.2.2 Different Rule Model

This verification is similar but different instances of `Rule Model'` are tested to verify if it is still possible to execute the Blended Workflow and achieve a complete state and in which situations can't. In this case the `User Model` and `Blended Workflow` used are described previously and do not change for any instance of the `Rule Model'`.

I've made verifications for five different instances of `Rule Model'`. In the first `Rule Model'` the permission for the `Receptionist` role define the `patient_name` attribute was removed, making it impossible to achieve the complete state. The second is similar to the first but instead of a definition permission, a read permission was removed also making it impossible to achieve the complete state. For the third instance of `Rule Model'`, the permission were changed from resources to operations and for the forth

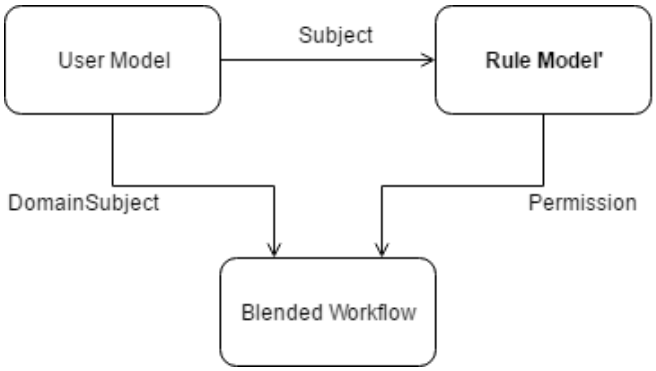

Figure 7.6: Different Rule Models Approach

the permissions were changed being an activity with permission over the operation and another over the resources. For these two verifications the assert generated a counterexample as expected meaning that both permission definitions can achieve a complete state.

Listing 7.8: Rule Model Modification

```
1  pred ruleModelWithoutARead{
2      AccessControlRules.resources = {
3          Patient +  patient_name + patient_episode +
4          Episode + episode_reserve_date + episode_patient +
5          registerPatient + createPatient + namePatient + bookAppointment + createPatientAndAppointment
6      }
7      AccessControlRules.permissions =
8          {Def -> {(R_Receptionist -> {Patient +  patient_name + patient_episode + Episode +
9                   episode_reserve_date + episode_patient})}
10         }
11     no AccessControlRules.delegation
12 }
13
14 assert differentRuleModelCantAchieveComplete2{
15     userModel and ruleModelWithoutARead and execution implies !complete
16 }
```

7.5.2.3 Different Blended Workflow Activities Specification

In this verification different variations of a `Blended Workflow'` are tested. These instances have the same resources of the original specification but have different operations which execution can achieve a complete state. Like the previous one the other two specifications: `User Model` and `Rule Model`, do not change.

The first instance of `Blended Workflow'` has the Register Patient activity divided in Create Patient and Name Patient, and the other instance has the activities Register Patient and Book Appointment

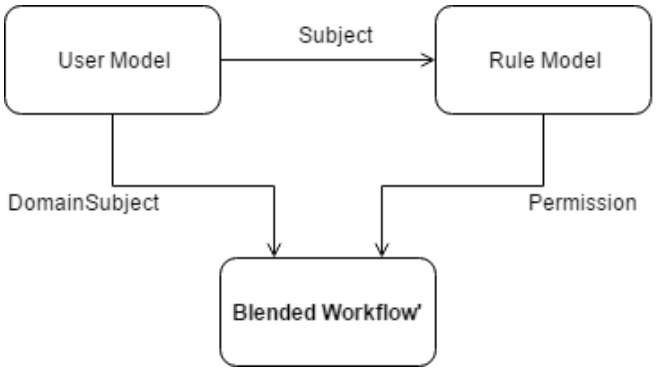

Figure 7.7: Different Blended Workflow Activity Specification Approach

merged in one. Given these specification I've tested them with three different instances of `Rule Model` in order to analyse the impact of having the permissions defined over operations. One with permissions over resources, other with half of the permissions over resources and the other half over operation and a third one only where the permissions only apply to the operations. From these three only the one with permissions over resources was able to maintain a execution that achieved a complete state.

Now I analyse what is the behaviour when an operation is split into two new operations. The previous instance of `Rule Model` that achieved the complete state for both instances of the `Blended Workflow'` had only one user with permissions to execute both activities. The `Rule Model` I am going to use has two users, the first user has a role that grants him the permission necessary over all resources of the activity Register Patient, and the second user has a different role which grant him permissions necessary over all resources of the activity Book Appointment. When I split the Register Patient activity into two activities, Create Patient which defines an instance of the `Patient` entity, and Name Patient which defines the *patient_name* of the `Patient`, it is not possible to achieve a complete state, because, it is now necessary to have a permission to read a patient when its name is being defined. From this case we can conclude that the split of activities may, in some situations, impede the complete state to be achieved.

The verification with the instance of `Blended Workflow'` that contains a merge of operations may also impede the achievement of a complete state. Although permissions exist to define the resources, for each one of the users, they to not have alone all the permissions, and so, none of them can execute the merged activity. A solution to this situation is to create a rule stating that the activity resulting from the merge could be propagated from one of the users to the other one. This way the user to which the permission is propagated can execute the activity if the other user allows it.

Listing 7.9: Rule Model Modification

```
1 pred newUserModel{
2       AccessControlRules.users = {Carl + John}
3       AccessControlRules.roles = {R_Receptionist + R_Receptionist2}
```

```
4       AccessControlRules.u_roles = {Carl -> R_Receptionist + John -> R_Receptionist2}
5   }
6
7   pred newRuleModel{
8       AccessControlRules.resources = {
9           Patient +  patient_name + patient_episode +
10          Episode + episode_reserve_date + episode_patient +
11          registerPatient + createPatient + namePatient + bookAppointment + createPatientAndAppointment
12      }
13      AccessControlRules.permissions =
14          {Def -> {(R_Receptionist -> {Patient +  patient_name })
15                  + (R_Receptionist2 -> {patient_episode + Episode + episode_reserve_date
16                  + episode_patient})}
17          + Read -> {(R_Receptionist2->{Patient})}}
18      no AccessControlRules.delegation
19  }
20
21  assert differentExecutionCantAchieveComplete7{
22      newUserModel and newRuleModel and executionWithOperationDivision
23          => !complete
24  }
25
26  assert differentExecutionCantAchieveComplete8{
27      newUserModel and newRuleModel and executionWithOperationMerge
28          => !complete
29  }
```

8

Conclusion

Contents

8.1 Conclusions

In this work I proposed and validated, through the use of Alloy language,an access control specification for the Blended Workflow which is comprehensive enough and covers most of the real life application cases. This access control uses most of the common requirements of access control such as user based access control, role based access control, domain association permissions, permissions over operations instead of resources and privileges propagation. The access control specifications are validated in order to find incoherences in the access control such as not being able to achieve the Blended Workflow instance goals due to the lack of permissions to execute some activities.

The validation done in this work allows a better comprehension of the definition of rules and permissions that are associated to basic resources in opposition to associating them to operations and the impact on the workflow execution when the definition of operations is changed. Although the definition of permissions with operations allows the access control to have far less rules it does not have the flexibility to support changes in the way resources are grouped in operations.

The work in this dissertation allowed to identify and validate the foundations of the access control semantics in the blended workflow, which will serve as a basis to its implementation in the blended workflow designer and engine.

8.2 System Limitations and Future Work

This work contains a few identified limitations. The first limitation is the scope used in Alloy execution and assertions. In the largest example used, the execution lasted between a minute and half and two minutes. Although this is not a huge amount of time, the example only has three entities, ten attributes and five bidirectional associations, and the execution only created an instance of each one. For a larger example the execution time would exponentially increase with the number of resources and scope. The other limitation is the lack of a user friendly interface which allows an end user without a technical knowledge to use the Alloy to specify and validate his access control. This specified access control only allows the definition of all rules and permissions in design time and does not address the modification of the control access rules during execution of workflow instances.

The next step of this work would be to implement the access control in the Blended Workflow tool. This implementation would allow the Blended Workflow specification and its access control to be extracted automatically from the data model and access control defined in the Blended Workflow tool eliminating the need for the end user to interact directly with the Alloy tool.

Bibliography

[1] A. Rito Silva, "A blended workflow approach," in *Business Process Management Workshops*. Springer, 2012, pp. 25–36.

[2] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers *et al.*, *Fundamentals of business process management*. Springer, 2013, vol. 1.

[3] A. Rito Silva and V. García-Díaz, "Blended workflow designer," Business Process Management Demonstration Track, 2016.

[4] D. Jackson, *Software Abstractions: logic, language, and analysis*. MIT press, 2012.

[5] S. Wu, A. Sheth, J. Miller, and Z. Luo, "Authorization and access control of application data in workflow systems," *Journal of Intelligent Information Systems*, vol. 18, no. 1, pp. 71–94, 2002.

[6] J. P. Near and D. Jackson, "Finding security bugs in web applications using a catalog of access control patterns," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 947–958.

[7] K. Andrews, S. Steinau, and M. Reichert, "Enabling fine-grained access control in flexible distributed object-aware process management systems," in *Enterprise Distributed Object Computing Conference (EDOC), 2017 IEEE 21st International*. IEEE, 2017, pp. 143–152.

[8] V. Künzle and M. Reichert, "Philharmonicflows: towards a framework for object-aware process management," *Journal of Software: Evolution and Process*, vol. 23, no. 4, pp. 205–244, 2011.