Visual Specifications of Policies and Their Verification*

Manuel Koch[1] and Francesco Parisi-Presicce[2,3]

[1] Freie Universität Berlin, Berlin (DE)
mkoch@inf.fu-berlin.de
[2] Univ. di Roma La Sapienza, Rome (IT)
parisi@dsi.uniroma1.it
[3] George Mason University, Fairfax VA (USA)
fparisi@ise.gmu.edu

Abstract. The specification of policies is a crucial aspect in the development of complex systems, since policies control the system's behavior. In order to predict a possibly incorrect behavior of the system, it is necessary to have a precise specification of the policy, better if described in an intuitive formalism. We propose policy specifications in three modeling notations, viz. UML, Alloy and Graph Transformations, and compare them from the viewpoint of readability, verifiability as well as tool support. We use a role-based access control policy as example policy.

1 Introduction

Policies are used to control the behavior of complex systems by a set of policy rules. Policy specifications are developed by system designers and deployed by administrators. To prevent an incorrect behavior of the system due to design or administration errors, a policy specification should be readable and understandable by both the designer during system development and the administrator who must deploy the policy. Visual modeling notations are designed to enhance the understandability of system aspects. The UML [13] as the de-facto standard modeling language in industry is a widely known member. Beside a readable policy specification, a formal specification of the policy is necessary to reason about the behavior of a policy and is a prerequisite for an effective analysis of conflicts within and between policies and their resolution to achieve consistency.

The aim of our investigation is to provide support for the design of a policy by proposing an intuitive visual representation along with a formal theory to develop and modify a policy in a systematic way. We investigate in this article to what extent the modeling notations UML, Graph Transformations [17] and Alloy [20] are suited for the specification and verification of policies. UML, as the standard modeling language in industry accompanied with several tools, is considered by showing how a policy can be specified in UML in such a way that we can make use of existing UML tools. We propose a formal graph-based

* Partially supported by the EC under Research and Training Network SeGraVis.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 278293, 2003.

semantics for the UML policy specification to make possible verification. Alloy, a lightweight object modeling notation, can express a useful range of structural properties in object models which can be automatically analyzed. Alloy provides a visual notation for parts of an object model. Graph Transformations (GT) are a graphical and formal specification technique, which incorporates both an intuitive visual notation and a formal background. Several works have reported on the use of GT for the specification of access control policies [8,9].

The paper is organized as follows. Section 2 investigates the necessary components of a policy specification and introduces the role-based access control policy example [18] used throughout the paper. Section 3 presents our proposal for a policy specification in UML, Sect. 4 the GT specification and Sect. 5 the Alloy specification. In Sect. 6, we investigate to what extent the specifications can be used to reason about the consistency of a policy specification and in Sect. 7 we compare the policy specifications from the point of view of readability, verifiability and tool support. Section 8 contains concluding remarks and future work.

2 Policies

Policies are employed to control the behavior of complex systems by using a set of policy rules that define the choices in the individual and collective behavior of the entities in the system. Beside the policy rules, declarative policy constraints may provide additional useful information on the intended behavior so that a policy specification contains also declarative information ("invariants") on what a system state must contain (positive) and what it cannot contain (negative). The declarative constraints provide useful information during the development of a policy through successive refinement steps, or when trying to predict the behavior of a policy. Therefore, a policy specification consists of 1) the type information of the system entities to which the policy applies, 2) a set of policy rules which build the accepted system states, and 3) a set of positive and negative declarative policy constraints for the wanted and unwanted substates.

2.1 Example: Role-Based Access Control

Role-based access control (RBAC) [18] reduces the complexity and cost of security administration in large systems because roles serve as a link between permissions (e.g., read or write) for objects (e.g., a file, a printer) and users. A user can access an object if (s)he plays a role which has the required permissions. The work on RBAC systems has been subject to a NIST standard proposal [19] which characterizes a family of RBAC models. The RBAC model in this article includes a role hierarchy, defining a sub-role (resp. super-role) relation between roles, whereby roles acquire the permissions of their sub-roles, and sub-roles acquire the user membership of their super-roles. The role hierarchy can be an arbitrary partial order and is assumed to be fixed. The RBAC policy rules are:

Rule 1: A user can be assigned to a role to become a member of the role. The user is authorized for a role, if this role is a sub-role of the role to which the user is assigned.

Rule 2: A user u can be revoked from a role r. We consider here only weak revocation, which revokes u only from this role r. Weak revocation has the property that a user u after revocation from r does not have to loose the permissions of r if u is assigned to a super-role of r, since super-roles inherit the permissions of their sub-roles. In strong revocation, the user u would be additionally revoked from all the super-roles of r.

Rule 3: A user can establish a session.

Rule 4: A user can close a session.

Rule 5: During a session of a user, the user can activate a subset of roles for which (s)he is authorized.

Rule 6: The user of a session can deactivate roles from the session.

The role-permission assignment [3] is assumed to be fixed. The following constraints are examples of additional restrictions of the RBAC model.

Separation of Duty (sod): This relation on roles places constraints on the assignments of users to roles. Membership in one role prevents the user from being a member of one or more other roles, depending on the sod-rules.

Cardinalities: This kind of constraint restricts the number of user-role assignments, user-session assignments etc. For example, a session must belong to a unique user.

The next sections present how the system entity types, the policy rules and the policy constraints are specified in UML, in Alloy and in Graph Transformations, exemplified with the RBAC model.

3 UML Specification

The type information of the system entities is specified in a UML class diagram. The class diagram for the RBAC example consists of the class *Role* for RBAC roles, *User* for users and *Session* for sessions (see Fig. 1a). An association shows, which class instances can be related. The label represents the intended meaning assigned to the association, its direction the direction to read the label (e.g., a *session belongs* to a *user*). The association *sod* on class *Role* specifies the separation of duty relation on roles, the association *is* represents the role hierarchy. A role r can be a super- or sub-role with respect to a role r'. Each role has a unique super-role, but a role can have zero or more sub-roles. If there is no multiplicity attached to an association, we assume the multiplicity $0..*$. The association *is_in* models the user-role assignment, the association *has_activated* models the roles that are activated in a session. The association *belongs* specifies the assignment of a session to a user.

The policy rules are specified in object diagrams using the constraints *new, all, destroy* and *destroy all* based on the constraints *new* and *destroyed* used in collaboration diagrams [13]. The intended meaning of the constraint *all* at an object in a policy rule is that all objects of this type must be considered which

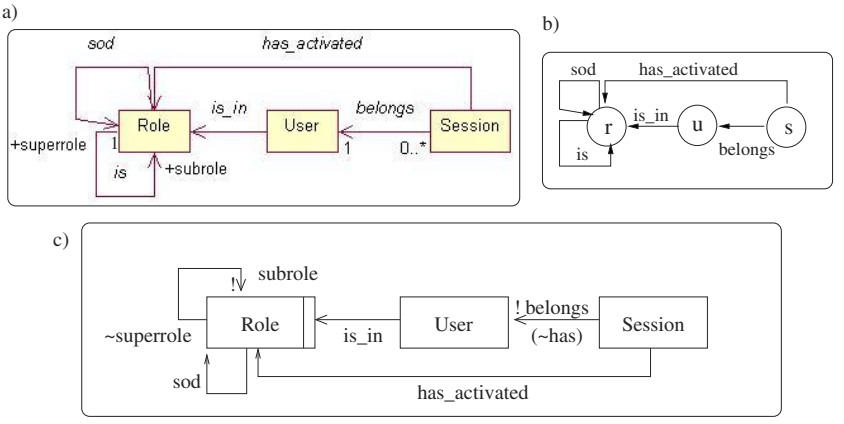

Fig. 1. Visual modeling of the type information in UML a), in GT b) and in Alloy c)

match to the object structure specified in the object diagram. An object or link with *new* is created by the policy rule, an object or link with *destroy* is removed from the system. The constraint *destroy all* at an object/link specifies that the policy rule deletes all objects/links that match the object structure specified in the object diagram. The match must be complete in the sense, that all mapped objects can be completed by links to achieve the object diagram in the rule.

Figure 2 shows the UML object diagrams for the policy rules of the RBAC model. Object diagram 1) consists of one object of type *User* carrying a constraint *new*. This specifies the creation of a user. Diagram 2) shows a user object connected to a session object. The constraint *destroy* at the user object specifies that the policy rule deletes the user, the constraints *destroy all* at the session object and the link specify that all sessions of the user are deleted, as well. Note, that sessions which are not connected to the deleted user object remain unchanged. Diagram 3) specifies the creation of a new session object connected to a user object. The user object already exists, only the session object and the link to the user are added by the policy rule. Diagram 4) specifies the deletion of a session and the connection to the user, diagram 5) the assignment of an existing user to an existing role by adding a new link. The revocation of a user from a role is specified in diagram 6). If a *user* is revoked from *role* (specified by destroying the link between the objects *user* and *role*), *user* may loose the authorization for all sub-roles of *role*. Therefore, all sub-roles of *role* must be deactivated from the sessions of the user, as well. The $*$ attached to the link between the roles *role* and *role'* models a path through the role hierarchy from *role* to *role'*. Since UML has no primitive operation for a transitive closure, transitivity must be specified in OCL. Therefore, we introduce the operation `closure()` on roles:

```
class Role
  closure(): Set(Role) =
    subrole.closure()->asCollection->including(self)
```

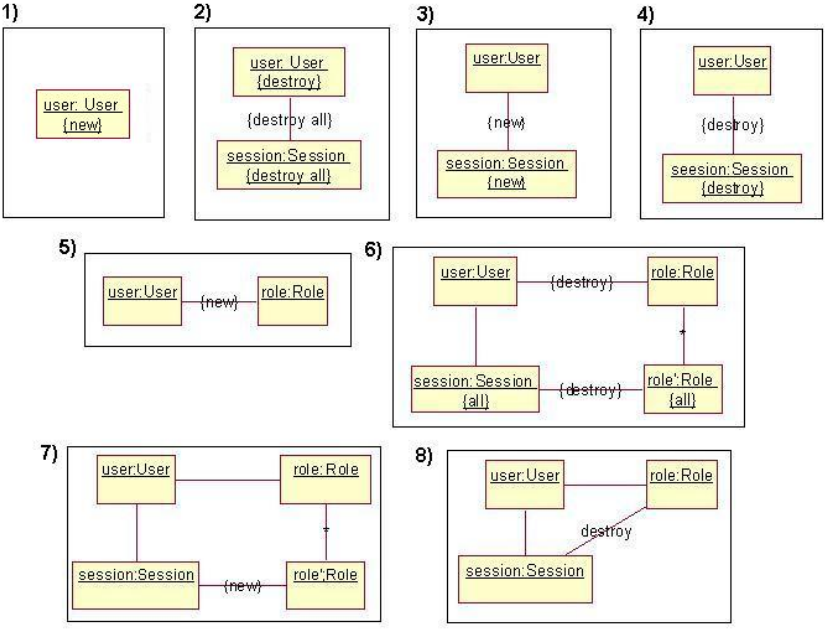

Fig. 2. UML object diagrams for policy rules

The *-link between objects *role* and *role'* in diagram 6) is a graphical notation for *role'* in *role.closure()*. Since we assume a non-cyclic role hierarchy, the operation `closure()` always terminates. Diagram 7) specifies the activation of a role in a user session by creating a new link between the session object and a role for which the user is authorized. A user is authorized for a role $role'$ if there is path starting from a role to which the user is assigned ending in $role'$ (specified by the *-link). Diagram 8) specifies the deactivation of a role from a session by destroying the link between the session object and the role object.

To specify policy constraints, we have chosen OCL [21]. The OCL constraint below specifies the separation of duty (sod) constraint.

```
context User inv
   self.is_in->forAll( r1,r2 | r1.sod->excludes(r2) )
```

The RBAC cardinality constraint, which requires a unique user for each session, is already specified by the multiplicity 1 at the association *belongs* in the class diagram in Fig. 1 a).

4 Graph Transformations Specification

Graph Transformations (GT) [17] provide an intuitive presentation of graph-based structures and their rule-based modification. The types of the system

entities are specified in a *type graph*. A type graph is a graph consisting of a set of nodes and a set of directed edges which specify the types of nodes and edges which may be used in the instance graphs modeling system states. A graph G is an instance graph of a type graph if one can find for each node and edge in G the corresponding node and edge type in the type graph. We take a GT approach in which edges are relations, i.e., there is at most one edge of the same type between two nodes, but there can be several edges of different type between the same nodes. Fig. 1b shows the type graph for the RBAC policy model. It has the types u for user, r for roles and s for sessions. The meaning of the edges corresponds to the meaning in the UML class diagram in Fig. 1a. In contrast to class diagrams, however, it is not possible to specify multiplicities in a type graph. An edge in a type graph always has the multiplicity $*$. Furthermore, edges in the type graph do not carry role names as associations in UML class diagrams.

The policy rules are specified by *graph rules*. Formally, a graph rule is given by a *graph morphism* $r : L \to R$ which consists of two partial injective mappings: one between the set of nodes and one between the set of edges of L and R, so that 1) whenever the mapping for edges is defined for an edge e pointing from node s to node t, the mapping for s and t is defined and the edge $r(e)$ in R points from $r(s)$ to $r(t)$ and 2) node/edges are mapped only to nodes/edges of the same type. We call the graph morphism *total* if the mappings between the node and edge sets are total. The graph L of a graph rule $r : L \to R$, *left-hand side* (LHS), describes the elements a graph must contain for r to be applicable. The morphism r is undefined on nodes/edges that are intended to be deleted, defined on nodes/edges that are intended to be preserved. Nodes and edges of R, *right-hand side* (RHS), without a pre-image are newly created. Note that the actual deletions/additions are performed on the graphs to which the rule is applied. The application of a rule to a graph G requires an occurrence of the LHS L in G (formally defined by the existence of a total graph morphism $m : L \to G$, called match). The application itself consists of two steps. First, delete all objects in G that have a pre-image in $L \setminus dom(r)$. Second, add all graph objects of $R \setminus r(L)$ to G connected to the nodes $m(dom(r))$.

The graph rules for the RBAC model are given in Fig. 3. The LHS of the graph rule *add user* is empty, its RHS contains one user node. This rule specifies the creation of a new user. The rule *remove user* deletes a user and all its connected sessions. The double circle around the session node specifies that all sessions connected to the user are deleted. The rule *new session* creates a new session for a user by inserting a new session node connected to the user node. The LHS of the graph rule *remove session* consists of the session node connected to the user node, the RHS consists only of the user node. This graph rule specifies the deletion of the session. The rule *add to role* assigns a user to a role. The rule *remove from role* specifies the revocation of a user from a role. The edge between the user and the role is deleted as well as all edges between sessions and roles for which the user is authorized by the revoked role. The * specifies a path between the roles. The rule *activate role* inserts an edge between a session of a user and a role for which the user is authorized. The graph rule *deactivate role*

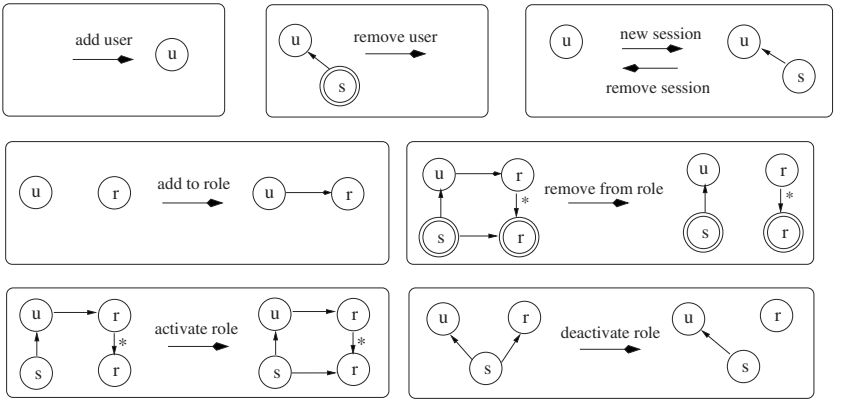

Fig. 3. Graph transformation rules for policy rules

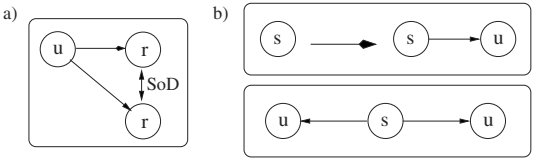

Fig. 4. Graphical constraints for policy constraints

deactivates a role from a session of a user by deleting the edge between session and role node.

Policy constraints are specified by *positive* and *negative graphical constraints* [10]. A positive graphical constraint (PGC) is a total graph morphism $c : X \to Y$ and a graph G satisfies c if for all total morphisms $p : X \to G$ there is a total morphism $q : Y \to G$ so that $q \circ c = p$. A negative graphical constraint (NGC) is a graph C and a graph G satisfies C if there does not exist a total morphism $p : C \to G$. The NGC in Fig. 4 a) specifies the separation of duty constraint. The NGC shows the forbidden substate in which a user is assigned to two roles in sod-relation. The two graphical constraints in b) specify that each session belongs to a unique user. The PGC requires for each session at least one connected user, the NGC forbids two (or more) users connected to one session.

5 Alloy Specification

Alloy[6] is a language for describing structural properties of object models and provides constraints and operations to describe how object structures may change. It is a state-based language with a textual syntax together with a graphical sublanguage. Alloy allows the designer to automatically analyze specifications [7]. The RBAC model of this paper is a submodel of the model used in [20].

The types for the system entities are specified in the Alloy diagram in Fig. 1c. Each box in the diagram represents a set of objects. We have the sets *Role*, *User* and *Session*. A vertical stripe down the right-hand side of the box specifies a fixed set, i.e., the number of elements in this set is fixed throughout the lifetime of the system. In the RBAC example, the roles are fixed, but the sets of users and sessions are not fixed. An edge represents a relation: for example, the relation *sub-role* maps a role to all its sub-roles, the relation *super-role* is defined to be its inverse (specified by the tilde). The relation *super-role* maps each role to its unique super-role. The exclamation mark attached to the end of a relation means exactly one. Another example is the relation *belongs* which maps each session to a unique user and its inverse *has* maps each user to his/her activated sessions.

Policy rules are specified textually by Alloy operations (keyword `op`). The primed values in operations indicate the post state of the variables. For sets, there are the usual set-theoretic operations for union (+) and difference (-), the operator `s in t` checks if the set `s` is a subset of `t`. For a relation `r`, the operation $\sim$`r` is the inverse of `r` and $*$`r` is the reflexive transitive closure of `r`. The '.' operator is used to specify a navigation expression, i.e., `s.r` denotes the set of objects that the set `s` maps to in the relation `r`. For example, `user.is_in` specifies the set of roles the `user` plays in a state.

```
op NewUser(user:User□) { User□ = User + user }

op RemoveUser(user:User□) {
  Session□ = Session - user.□has
  User□ = User - user
}
op NewSession(user:User, session:Session□) {
  Session□ = Session + session
  session.belongs□ = session.belongs + user
}
op RemoveSession(user:User, session:Session□) {
  user.□has□ =  user.□has - session
  Session□ = Session - session
}
op AddToRole( user:User, role:Role ) {
  user.is_in□ = user.is_in + role
}
op RemoveFromRole(user:User, role:Role) {
   user.is_in□ = user.is_in - role
   user.□has.is_activated□=user.□has.is_activated-role.*subrole
}
op ActivateRole(user:User,session:Session,role:Role,role2:Role){
   role in user.is_in
   role2 in role.*subrole
   session.is_activated□ = session.is_activated + role2
}
```

```
op DeactivateRole(user:User, session:Session, role:Role) {
   session.is_activated' = session.is_activated - role
}
```

Policy constraints are described by Alloy assertions (keyword `assert`), which are questions of the kind "Is it true that ...?". The Alloy analyzer tries to answer this question by finding a counterexample (more in Sect. 6). The assertion for the sod-constraint states that when a user u is assigned to roles $r1$ and $r2$ then $r1$ and $r2$ are not in sod-relation (`->` specifies implication, `!` negation).

```
assert SeperationOfDuty{ all r1,r2: Role, u:User |
                         (r1 + r2) in u.is_in -> r1 != r2.sod }
```

The cardinality constraint of a unique user for each session is visually specified by the exclamation mark attached to the relation *belongs* in the Alloy diagram.

6 Verification

A crucial property of a policy specification is that it specifies a coherent policy in the sense that the policy rules and the policy constraints are not contradictory. Therefore, we define a policy specification to be *coherent* if all system states built by the policy rules satisfy the constraints. In the context of UML, a system state is an object model and a policy constraint an OCL constraint. Satisfaction deals with the satisfaction of the OCL constraint in the object model. In the context of Alloy, a system state is an instance of the Alloy model with concrete instance sets and relations. A policy constraint is an Alloy assertion and satisfaction deals with the satisfaction of the assertion in the instance model. In the context of GT, a system state is a graph and a policy constraint is a graphical constraint. Satisfaction deals with satisfaction of the graphical constraint by the state graph.

The RBAC policy specifications in the previous sections are not coherent, since the sod-constraint is not satisfied by all system states built by the policy rules. The policy rule for the assignment of a user to a role (diagram 5 in Fig. 2 in the case of UML, graph rule *add to role* in Fig. 3 in the case of GT, operation `AddToRole` in the case of Alloy) does not consider the sod-relation. The rule can assign a user to any role and may construct a system state which does not satisfy the sod-constraint. This section investigates the problem of detecting incoherence in a policy specification and, if so, of resolving it.

6.1 Alloy

Alloy is supported by an analyzer [7] to detect constraint violations. The analyzer checks Alloy assertions by trying to find a counterexample, i.e., to find an instance model of the Alloy specification in which the assertion is not satisfied. Using the Alloy analyzer to check the sod assertion of the policy specification in Sect. 5, the analyzer gives the counterexample in Fig. 5. The $User1$ is assigned to the roles $Role2$ and $Role3$ which are in sod-relation.

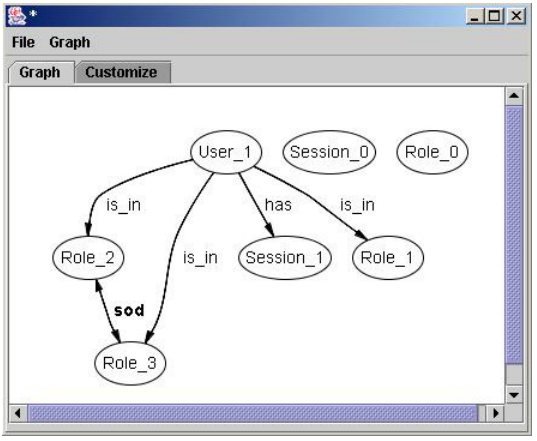

Fig. 5. Counterexample found by the Alloy analyzer

Based on the counterexample, the designer tries to modify the Alloy specification to resolve the conflict. In our example, the invariant `assign` is added to the Alloy specification. It specifies that roles $r1$ and $r2$ both assigned to a user u must not be in sod-relation. Applying the Alloy analyzer to the extended Alloy specification results in no solution, i.e., no counterexample could be found and the Alloy policy specification is coherent.

```
inv assign { all u: User, r1: Role, r2: Role |
             (r1 + r2) in u.is_in -> r1 !in r2.sod }
```

6.2 Graph Transformations

Graph Transformations provide a constructive approach to the detection of constraint violation [4,8]. The algorithm receives as input a graph rule and a graphical constraint and decides whether the graph rule may construct a graph which does not satisfy the graphical constraint. When a conflict is detected, the algorithm modifies the graph rule by adding a negative application condition (NAC) so that the modified graph rule is applicable only when it is guaranteed to produce a new graph that satisfies the graphical constraint. A NAC for a graph rule $r : L \to R$ is a total graph morphism $n : L \to N$ and r with NAC n can be applied to a graph G if there is a match $m : L \to G$, but there is no total morphism $q : N \to G$ so that $q \circ n = m$.

Applying the algorithm to the graph rule *add to role* in Fig. 3 and the graphical constraint for sod in Fig. 4, the algorithm detects that the rule may violate the constraint. Therefore, the algorithm adds a NAC to the rule (Fig 6 a)). In the representation of a graph rule with NAC, the nodes and edges of L are drawn by solid lines, the part $N \setminus n(L)$ by dotted lines. The NAC for rule *add to role* is given by the dotted role node r', the dotted edge between the user u and role

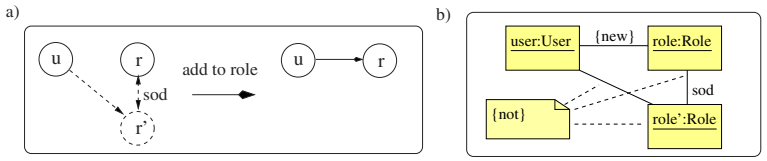

Fig. 6. Consistent graph rule *add to role* and its translation into a UML policy rule b)

r' as well as the dotted sod-edge. The NAC forbids the assignment of a user to a role r if the user is assigned to a role r' which is in sod-relation with r.

6.3 UML – A Graph-Based Formal Semantics

There is some tool support for OCL constraints ranging from syntactical analysis, type checking, dynamic invariant validation, dynamic pre-/post-condition validation or test automatization [16,5]. Since our aim is to check the coherence of a policy specification in UML, we focus on the validation of invariants supported, for example, by the USE tool [15,14]. The USE tool allows the designer to validate OCL constraints against snapshots of the system. To check the coherence of a policy specification, however, we have to consider all system states that can be built by the policy rules. Since it is not possible to generate all these states by the tool, it is not suitable to check the coherence of a policy.

Therefore, we propose a translation from the UML policy specification into a GT specification, based on the results in [11]. The translation of a class diagram into a type graph is essentially straightforward: each class x becomes a node x, each directed association a becomes an edge a (for undirected associations a we get two edges pointing in opposite direction). The multiplicities in a class diagram are converted into graphical constraints. A multiplicity range $n...m (n < m)$ for an association is translated into a positive graphical constraint (PGC) for the lower bound n and a negative graphical constraint (NGC) for the upper bound m. The graph X of the PGC $c : X \to Y$ for the lower bound n contains one object node of the association's source class. The graph Y contains the same node and n object nodes for the association's target class. The morphism c maps the object node in X to its counterpart in Y. The graph C of the NGC for the upper bound m contains one object node of the association's source class and $m + 1$ objects of the association's target class. For example, the multiplicity 1 at association *belongs* is translated into the graphical constraints in Fig. 4 b).

The UML object diagrams for policy rules are translated into graph rules $r : L \to R$, where L is the graph given by all objects/links without constraint *new* and R is the graph given by all objects without *destroy* or *destroy all*. Objects with constraint *all* or *destroy all* become a node with a double circle. Associations with attached $*$ are translated to edges carrying a $*$. The rule morphism r is defined for all objects/links without *new* and maps the objects/links to their counterparts in R. The graph rules in Fig. 3 are the result of the translation of

the UML policy rules in Fig. 2. The Fujaba system [12] implements the idea to use UML collaboration diagrams as a front-end notation for GT rules.

Whereas the translation of the class diagram and the policy rules can be done automatically due to a similar visual presentation, the translation of OCL constraints in graphical constraints must be done by the designer. In [1] an automatic translation of OCL constraints into graphical constraints is shown. But only simple OCL constraints can be translated. Another possibility is a direct visual specification of UML policy constraints as proposed in [11]. When the OCL constraints are translated into graphical constraints, the coherence of the policy specification can be checked by the algorithm in 6.2. To translate a graph rule with NAC into an object diagram for the UML policy rule, we attach a note with constraint *not* to all objects/links coming from the NAC. Fig. 6b shows the translated UML object diagram from the graph rule in a).

7 Comparison

Our aim is a policy specification framework that provides an intuitive visual notation usable by policy designers, deployers and administrators along with a formal theory to reason about the coherence of a policy specification. The framework would benefit from a tool support for both specification and verification. We compare to what extent the presented approaches bring forth this aim. Table 1 shows a summary of the comparison. Column *visual* states which parts of a policy can be specified visually, column *verification* states if there are concepts to check constraints or to resolve incoherence, column *tool* states the tool support for the specification and verification of policies.

Table 1.

	visual	verification	tool
UML	types, policy rules	no, but translation into GT possible	specification
Alloy	types	constraint checker	specification, verification
Graph Transformations	types, policy rules, policy constraints	constraint checker, conflict resolving	specification

7.1 Visual Specification

The visual specification of the type information of a policy is quite similar in all three approaches. The diagrams in Fig. 1 describe almost the same graph and differ only in small parts. For example, both the UML class diagram and the Alloy diagram provide the specification of multiplicities for associations/relations,

whereas the GT type graph specifies only the existence or non-existence of connections. Multiplicities in GT must be specified separately in graphical constraints, which is less intuitive than the direct presentation in the diagram.

The difference between the approaches becomes more significant when policy rules and policy constraints are specified. Graph Transformations rules provide an intuitive graphical notation to specify both the triggers of a policy rule, i.e., when to apply a policy rule, and the policy rules' effects in one compact representation. UML provides several possibilities which could be used to describe the policy rules. As an alternative to our proposal of object diagrams, sequence diagrams could be used, in which messages specify both the triggers of a policy rule and its effects. Messages may cause a change of the object structure. Since the order of the messages for policy rules is less interesting (the trigger is the first message, the message order for the policy rule effects is generally not important) than the change of the object structure, collaboration diagrams appear to be more suitable. In fact, the object diagrams chosen in our approach are the context of collaboration diagrams without any interaction. Our approach can therefore be easily extended, if interaction must be considered in policy rules, as well. The expressiveness of Alloy operations allows the designer to specify policy rules, but there is no graphical representation for Alloy operations (yet). It is possible, however, to provide also a visual notation to Alloy similar to the one proposed for UML.

Visual constraint specification is only possible in GT. Neither the OCL constraints nor the Alloy assertions/invariants can be presented visually. As mentioned above, simple OCL constraints, however, could be represented also graphically [1].

7.2 Verification

Given a policy specification, we want to determine if the policy rules may violate the policy constraints. The Alloy analyzer checks the constraints by looking for a counterexample, i.e., an instance model produced by the policy rules which does not satisfy the constraint. If a counterexample is found, however, the Alloy analyzer does not say which operation (i.e., policy rule) causes the inconsistency. The designer has to interpret the counterexample and change the Alloy specification for a resolution.

In GT, constraints and rules are checked pairwise to detect inconsistencies. When a conflict is detected, the algorithm gives the designer the graph rule and the graphical constraint which cause the conflict. Unlike with Alloy, the conflict is automatically solved by modifying the graph rule and maintaining the constraint.

OCL constraints can be checked only w.r.t. snapshots of the system, but there are no concepts in UML to check the coherence of a policy, i.e., whether the policy rules may build a system state in which an OCL constraint is not satisfied. As shown in 6.3, the UML specification can be translated into a GT policy specification to use GT verification concepts.

7.3 Tool Support

Tool support for the specification of a policy is partly available for all three notations. UML is accompanied by several CASE tools and our approach to the UML policy specification can be written in existing UML tools, since only standard UML extension mechanisms are used. The Alloy policy specification can be written in the Alloy analyzer tool [7]. The specification for the analyzer, however, is completely textual and does not support the graphical notation presented in Sect. 5. For the specification of the GT policy specification, general GT tools can be used [2].

Tool support for the verification of the coherence of a policy is only available in the Alloy analyzer. There is no tool support for checking the coherence of a policy specification either in UML or in GT, since the algorithm for detecting and resolving constraint violations in GT is not yet implemented.

8 Concluding Remarks

To reduce errors in the behavior of complex systems due to faulty policy specifications or due to administration errors of correct, but incomprehensible, policy specifications, we have presented policy specifications in UML, Alloy and GT, providing a concise notation as visual as possible and a formal semantics to reason about policy coherence. We have proposed a way to express the policy components in UML so that, on the one hand, UML tools can be used and, on the other hand, a formal semantics based on GT can help to reason about the coherence of a policy. Future work could investigate the translation of a UML policy into an Alloy policy to use the Alloy analyzer in the UML context.

None of the notations, however, reaches our aim completely. In Alloy and UML, not all policy components, especially constraints, can be expressed visually. This is possible in GT, but the tool support to check the policy is missing, even if the theoretical results do exist. The tool support for the verification is an advantage of Alloy. Future work will deal with a tool which implements the GT checking algorithm and the automatic translation of an XMI representation of a UML policy specification into a GT representation. This allows the designer to import UML policies and to check their coherence by means of the GT checking algorithm.

To get practical results about the actual understandability, the proposed notations must be further evaluated (preferably) by user tests. Moreover, these tests would help to refine the proposed notations to missing concepts needed in policy specifications or to adapt the notations to special user requirements. For example, the UML notation of this paper is one possibility to specify a policy in UML to which a GT semantics can be given. But there may be other UML representations (e.g., using statecharts/sequence diagrams) that are more convenient for software engineers and clients.

References

1. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In *Proc. UML2000*, number 1939 in LNCS, 2000.
2. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. II: Applications, Languages, and Tools*. World Scientific, 1999.
3. P.A. Epstein. Engineering of Role/Permission Assignments. *PhD Thesis, George Mason University*, 2002.
4. R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In *Proc. SEGRAGRA'95 Graph Rewriting and Computation*, number 2. Electronic Notes of TCS, 1995. `http://www.elsevier.nl/locate/entcs/volume2.html`.
5. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *Proc. of UML2000*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.
6. D. Jackson. Alloy: A Lightwight Object Modelling Notation. Technical Report 797, MIT Laboratory for Computer Science, 2001.
7. D. Jackson, I. Schlechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, 2000.
8. M. Koch, L.V. Mancini, and F. Parisi-Presicce. A Graph Based Formalism for RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):332–365, August 2002.
9. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the Specification of Access Control Policies. In F.Honsell and M.Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, Lect. Notes in Comp. Sci. Springer, March 2001.
10. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Conflict Detection and Resolution in Access Control Specifications. In M.Nielsen and U.Engberg, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2002)*, Lect. Notes in Comp. Sci., pages 223–237. Springer, 2002.
11. M. Koch and F. Parisi-Presicce. Access Control Policy Specification in UML. In *Proc. of UML2002 Workshop on Critical Systems Development with UML*, number TUM-I0208, pages 63–78. Technical University of Munich, September 2002.
12. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. *Proc. of the 22nd Int. Conf. on Software Engineering*, 2000.
13. OMG. OMG Unified Modeling Language Specification, V.1.4, 2001.
14. M. Richters. The USE tool: A UML-based specification environment, 2001. `http://www.db.informatik.uni-bremen.de/projects/USE`.
15. M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In *Proc. UML2000*, 2000.
16. M. Richters and M. Gogolla. OCL – Syntax, Semantics and Tools. In *Proc. Advances in Object Modelling with OCL*, LNCS, pages 38–63. Springer, 2001.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*. World Scientific, 1997.
18. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. In *1st ACM Workshop on Role-based access control*, 1996.
19. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proc. of the 5th ACM Workshop on Role-Based Access Control*. ACM, July 2000.

20. A. Schaad and J.D. Moffett. A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*. ACM Press, 2002.
21. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.