RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis

John Zao, Hoetech Wee, Jonathan Chu, Daniel Jackson

Abstract — Assuring logical consistency among the specification of entities, relations and constraints in a role-based access control (RBAC) schema becomes increasingly important and urgent as the access control systems increase their spatial and temporal span. In this paper, we report an attempt to employ ALLOY, a lightweight modeling system with automatic semantic analysis capability, to verify internal consistency of RBAC schema as well as explicit manifestation of certain algebraic properties.

1. INTRODUCTION

Access control is used extensively in information systems as a security mechanism for protecting sensitive information and resources from illegitimate access. Because correct specification and implementation of access control rules — known as *access control schema* or *policies* — are crucial to the correct functioning of access control systems, it is impertinent that we analyze the access control schema and verify the correctness of their implementation before putting them into actual use. A few classical access control schema mainly variations of Bell-LaPadula [BLP75] and China Wall [BN89] models have been carefully studied. However, these idealized models often fail to take into account the peculiarity of real-life scenarios. Hence, the development of access control schema for practical applications remains an art relying largely on experience and intuition.

With the advent of client-server systems and the Internet, access control becomes a critical means for protecting information and resources available on the Internet. In addition to traditional *mandatory* and *discretionary access control*, a new paradigm known as *role-based access control (RBAC)* [FK92] [SCFY96][GI96] has gained popularity among commercial vendors and users as a promising way to realize fine-grain access control of network databases. The essence of RBAC lies with the notion of *roles* as an intermediary between conventional access control subjects and objects: roles are given permissions to access objects while subjects are associated with roles. The introduction of roles greatly simplifies the management of access control systems as it separates the dynamic associations between subjects and roles from the relatively static associations between roles and permissions. It also enriches the access control schema model by introducing the concepts of role hierarchies and association constraints. Along with the flexibility of RBAC comes a vast management problem; the sheer number of roles and the dynamics of associations make verifying the correctness of a RBAC schema a crucial and yet difficult task. One small change to a permission assigned to a role may create security loopholes or introduce permission conflicts that can hardly be detected by manual inspection. Furthermore, roles and constraints are often added,

deleted or modified throughout the lifetime of a system by various personnel who have only partial knowledge of the system. All these concerns imply that a reliable method for checking the consistency among roles and constraints is desperately needed. Currently, the technology for automated verification of RBAC schema is not yet developed although standard methods for schema specification have been emerging; e.g., Ahn and Sandhu [AS00] developed the RCL-2000 language for specifying RBAC constraints. In a DARPA funded project, ALLOY-based Value and Dependence Abstraction, (AVDA) [Z+01], BBN developed a RBAC schema debugger prototype that uses a lightweight formal modeling system developed in MIT called ALLOY [Jac00b] to verify the algebraic properties and the logical consistencies of RBAC schema.

The RBAC Debugger uses a constraint analyzer built into the lightweight modeling system to search for inconsistencies between the mappings among users, roles, objects, permissions and the prohibitive[1] and the obligatory[2] constraints in a RBAC scheme. The debugger was demonstrated to have the following capabilities: (1) specifying roles, user-role and role-permission associations according to RBAC-96 schema framework [SCFY96], (2) specifies static role/user/permission-centric separation-of-duty (SoD) constraints as depicted in [AS00], (3) verifies consistency between user-role/role-permission associations and static SoD constraints, (4) verifies consistencies between user-role/role-permission associations and the algebraic properties of RBAC schema and (5) searches for a plausible realization of the RBAC schema. The ability of ALLOY to conduct *satisfiability analysis* [G+96] using a built-in Constraint Analyzer gives it an edge to surmount the challenge of automatic schema verification. In particular, ALLOY can be used to perform the following two tasks:

1. Develop an "abstract" access control schema and then verify the *correctness* of different RBAC implementations against the abstract schema;
2. Specifying the properties of RBAC entities such as subjects, objects, roles and their associations by asserting prohibitive and obligatory constraints, and then check the *consistency* between the existing constraints and the future changes to the RBAC entities and constraints.

In this paper, we report both our effort in developing the RBAC schema debugger and the results of our experiments with the debugger prototype. Next section contains a succinct summary of the principle and the features of ALLOY system. Section 3 provides the first example of ALLOY use as we specified Bell-

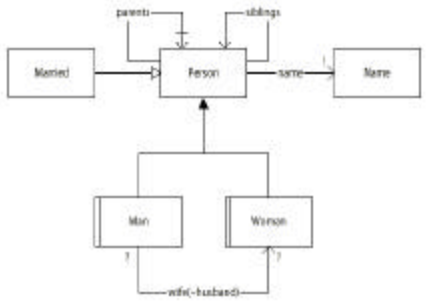
[1] *Prohibitive constraints* are the predicates specifying the instances that can never occur in a realization of the RBAC model.

[2] *Obligatory constraints* are the predicates specifying the conditions that must be satisfied by any realization of the RBAC model.

LaPadula confidentiality service in ALLOY modeling language and verify its read/write properties using ALLOY Constraint Analyzer. Section 4 gives the specification of entities and relations embodied in RBAC-96 framework. Section 5 and 6 demonstrate ALLOY capability in performing the two tasks mentioned above. Concluding comments are provided in Section 7.

2. ALLOY LIGHTWEIGHT MODELING SYSTEM

ALLOY [Jac99][Jac00a][Jac00b] is a textual notation designed for software system modeling. It is a small language — much smaller than any programming language, and smaller than most modeling languages. A subset of ALLOY can be expressed graphically. In addition to the standard set operators, it provides a 'navigation' operator that allows structural constraints to be succinctly expressed, and supports descriptions of evolution of system structures over time. Transitive closure is built-in, so there is no need for iteration constructs. ALLOY is strongly but implicitly typed, so many simple errors in models are caught without burdening the user with type declarations. By treating scalars as singleton sets, ALLOY sidesteps the problem of undefined expressions, and allows relations and functions to be treated uniformly. Figure 4 gives an example of an ALLOY model of human family relations. Note that ALLOY supports the definition of operators and their association with entities. Also, it supports the assertion of logical predicates, and its Constraint Analyzer can check the consistency of the assertion against the model.




```
model Family {
domain {Person, Name}
state {
    partition Man, Woman : static Person
    Married : Person
    parents : Person -> static Person
    siblings : Person -> Person
    wife (~husband) : Man ? -> Woman ?
    name : Person -> Name !
}
def siblings {
    all a, b | a in b.siblings <-> (a.parents = b.parents)
}
inv Basics {
    all p | some p.wife <-> p in Man & Married
    no p | p.wife / in p.siblings
    all p | (sole p.parents & Man) && (sole p.parents & Woman)
    no p | p in p.+parents
    all p, q | p.name = q.name -> no (p.parents & q.parents)
}
op Marry (m: Man!, w: Woman!) {
    m not in Married && w not in Married
    m.wife' = w
    all p: Man - m | p.wife' = p.wife
    all p | p.name' = p.name
    all p | p.parents' = p.parents
    Person' = Person
}
assert HusbandsWife {
    all p : Married & Woman | p.husband.wife = p
    }
}
```

Figure 1. ALLOY Model of Human Family Relations

2.1 ALLOY Logic System[3]

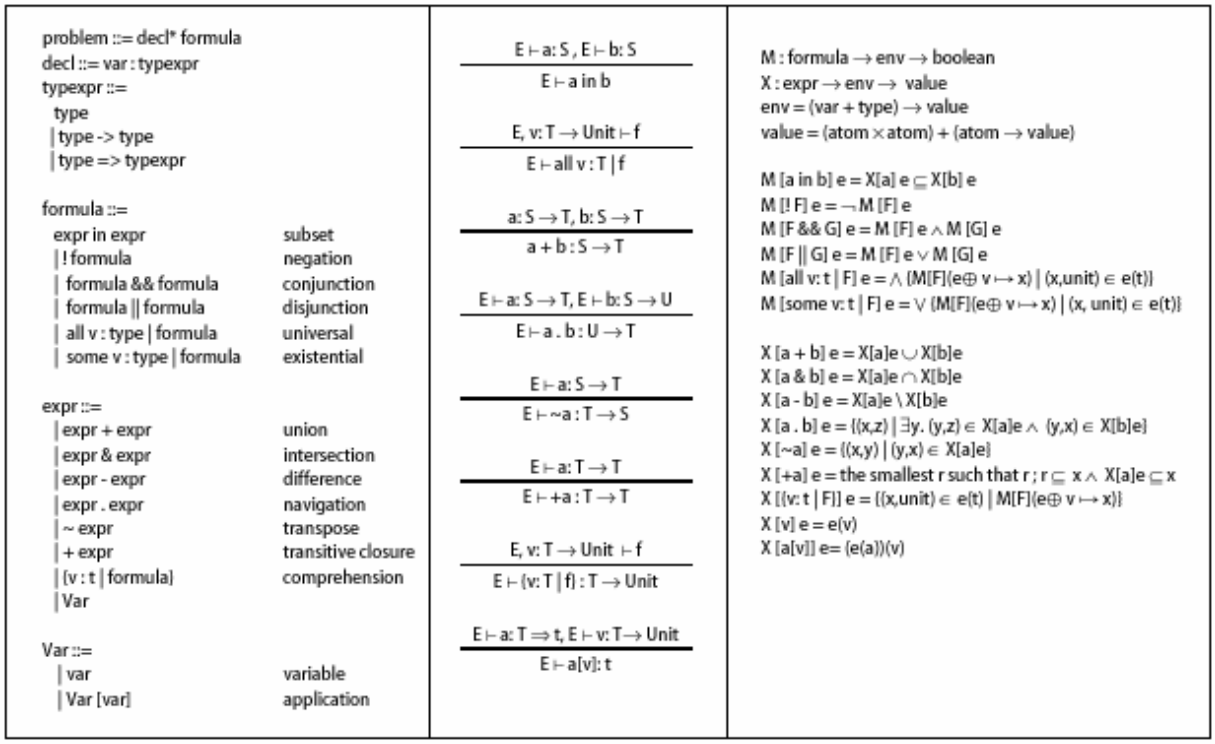


Figure 2. ALLOY Syntax, Type and Semantic Definitions

ALLOY was designed to be *lightweight, precise* and *tractable.* In other words, it should be (1) small and yet capable of expressing common properties of object-based system tersely, (2) formally defined with simple uniform mathematical semantics, and (3) amenable to efficient and automatic semantic analysis. Figure 2 gives the definition of ALLOY logic system with abstract syntax on the left, type system in the middle and semantics on the right. Following are a few remarks about the syntax and formal semantics of ALLOY.

The syntax of ALLOY is mostly identical to standard mathematical syntax of the first order logic, but uses ASCII rather than typographic symbols for the operators. The logic is strongly typed, and a formula is accompanied by declarations of the set and relation variables; we call the combination of a formula and its declarations a problem. Each declaration associates a type with a variable. There are three kinds of type:

- Set type T, denoting sets of atoms drawn from T;
- Relation type S -> T, denoting relations from S to T;
- Function type T => t, denoting functions from atoms of T to values of type t.

[3] This brief description of ALLOY syntax and semantics was extracted from [Jac00b].

Types are constructed from basic types that denote disjoint sets of atoms. Functions correspond to predicates of arity greater than two. There are no scalar types. To declare a scalar variable, we declare it to be a set $v : T$ and add a constraint that makes the set a singleton:

$$some\ x: T \mid x = v$$

This allows navigation expressions to be written uniformly, without the need to convert back and forth between scalars and sets, sidesteps the partial function problem, and simplifies the semantics and its implementation. Finally, expressions are formed using the standard set operators (union, intersection and difference), the unary relational operators (transpose and transitive closure), and the dot operator, that is used to form navigation expressions.

The semantics of ALLOY is defined by a standard denotational semantics with two meaning functions: M, which interprets a formula as true or false, and X, which interprets an expression as a value. Values are either binary relations over atoms or functions from atoms to values. Interpretation is always in the context of an environment that binds variables and basic types to values, so each meaning function takes both a syntactic object and an environment as arguments.

All operators have their standard interpretation, except the dot operator, which does double duty. Its semantic definition is like relational composition, but with one argument transposed and the arguments reversed. Our motivation for defining dot this way is that, when s is a set and r is a relation, s.r denotes the image of s under r. Combining this with the treatment of scalars as singleton sets results in a uniform syntax for navigation expressions. For example, if p is a person, p.mother will denote p's mother; p.parents will denote the set of p's parents; p.parents.brother will denote p's uncles; etc.

The environments for which the formula is true are called the *instances* or *solutions*[4] of the formula. If a formula has at least one solution, it is said to be *consistent*; when every well-formed environment is a model, the formula is *valid*. The negation of a valid formula is inconsistent, so to check an assertion, we look for a model to its negation; if one is found, it is a *counterexample*. Since the logic is undecidable, it is impossible to determine automatically whether a formula is valid or consistent. We therefore limit our analysis to a finite *scope* that bounds the sizes of the carrier sets of the basic types. We say that a model is *within a scope of* k if it assigns to each type a set consisting of no more than k elements. Clearly, if we succeed in finding a model to a formula, we have demonstrated that it is consistent. Failure to find a model within a given scope, however, does not prove that the formula is inconsistent (although in practice, for a large enough scope, it often strongly suggests it).

[4] They are also referred to as the *models* in formal logic.

2.2 ALLOY Constraint Analyzer

The ALLOY modeling system is equipped with the ALLOY Constraint Analyzer [JSS00], which can simulate ALLOY specifications, generate sample structures and transitions, and check user-defined properties. Presently, it is the only tool that can perform automatic semantic analysis of structural models. ALLOY is undecidable, so it is not possible to check arbitrary properties. Instead, the analyzer checks claims by looking for counterexamples. Using new satisfiability (SAT) solving technology, it can perform an exhaustive search of all structures involving a fixed number of atomic elements. Searches involving 10^{60} structures can usually be completed in seconds. Figure 3 shows a screen shot of ALLOY Constraint Analyzer in operation.

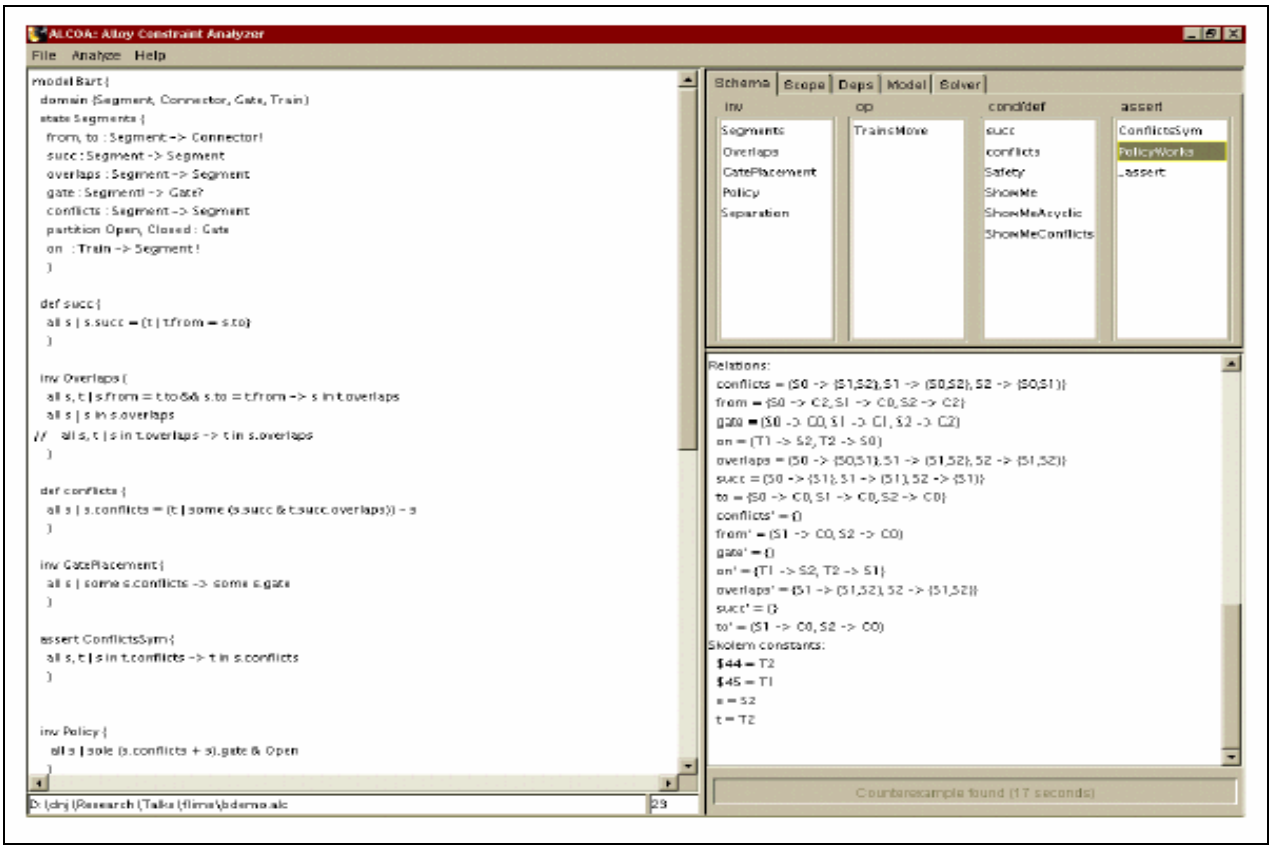


Figure 3. Sample display of ALLOY Constraint Analyzer

3. SPECIFICATION OF BELL-LAPADULA CONFIDENTIALITY SERVICE

As a demonstration of the expressive power of ALLOY and the analysis capability of its Constraint Analyzer, we used the formal modeling system to specify the read-down and write-up (*-property) rules of Bell-LaPadula confidentiality protection scheme [BLP75].

It is well known that classical access control schema such as Bell-LaPadula and China Wall are *lattice based* and can be specified in terms of partial order relations. Since ALLOY was built upon first-order logic, the modeling system permits its users to define mathematical relations by specifying their algebraic properties. The following ALLOY statements define *partial order* as a reflexive, anti-symmetric and transitive relation, and used it to establish a lattice structure among four security levels, Regular or Unclassified (R), Classified (C), Sercet (S) and Top Secret (TS). Figure 4 displays a graphic rendering of the lattice that was produced by ALLOY Constraint Analyzer.

```
state {
  partition R, C, S, TS: fixed Labels!
  leq: Labels -> Labels+  }
inv PartialOrderRelations {
  // reflexive
  all r1 | r1 in r1.leq
  // transitive
  all r1, r2 | r1 in r2.*leq -> r1 in r2.leq
  // anti-symmetric
  all r1, r2 |
  r1 in r2.leq && r2 in r1.leq -> r1 = r2  }
inv SecurityLevelOrder {
  C in R.leq
  S in C.leq
  TS in S.leq  }
```

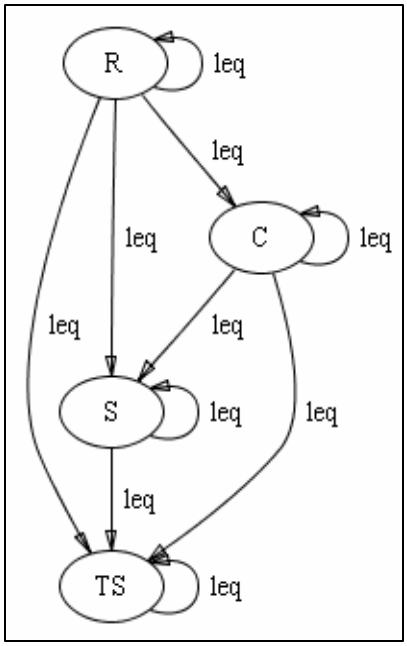


Figure 4. Partial-order relations among Bell-LaPadula security levels

Based on the lattice structure, we can easily express the read-down (`canRead`) and write-up (`canWrite`) rules as follow:

```
domain {
  fixed SecurityLabels,
  fixed Documents,
  fixed Users
}
state {
  secClass: Documents -> static SecurityLabels!
  secClearance: Users -> static SecurityLabels!
  leq: SecurityLabels -> SecurityLabels+
  canRead: Users -> Documents
  }
def canRead {
// read down property:
    all u | u.canRead = { d | u.secClearance in d.secClass.leq }
}
def canWrite {
// write up or * - property
   all u | u.canWrite = { d | u.secClearance in d.secClass.geq }
}
```

The analytical power of ALLOY was clear when we made the assertion that people with higher security clearances can read all the documents that can be read by those with lower clearances:

```
assert MorePowerfulCanReadMore {
  all u1, u2 |
    u2.secClearance in u1.secClearance.leq
     -> u1.canRead in u2.canRead
}
```

and then tested the validity of the assertion using the Constraint Analyzer. Similarly, we also made the assertion that the partial-order relations among the security levels and the read-down rules applicable to their subjects are equivalent:

```
    cond Equivalent {
           all u1, u2 | u2.secClearance in u1.secClearance.leq <-> u1.canRead in
u2.canRead
    }
```

4. SPECIFICATION OF RBAC-96 SCHEMA FRAMEWORK

ALLOY is a general-purpose modeling system. We must adapt it to the task of RBAC schema specification and verification by providing it with the definitions of RBAC entities including subjects (which can be subdivided into users and principals), roles and permissions (which are composed of objects and operations) as well as relations among different entities. These definitions should be written in ALLOY modeling language and loaded into the modeling system as a macro library. Significant amount of effort has been spent to develop the macro library based upon a common framework for RBAC schema specification [SCFY96]. Figure 5 shows the Entity-Relationship diagram of RBAC-96 Framework. Appendix A supplies the ALLOY definition of RBAC-96 entities and relations.

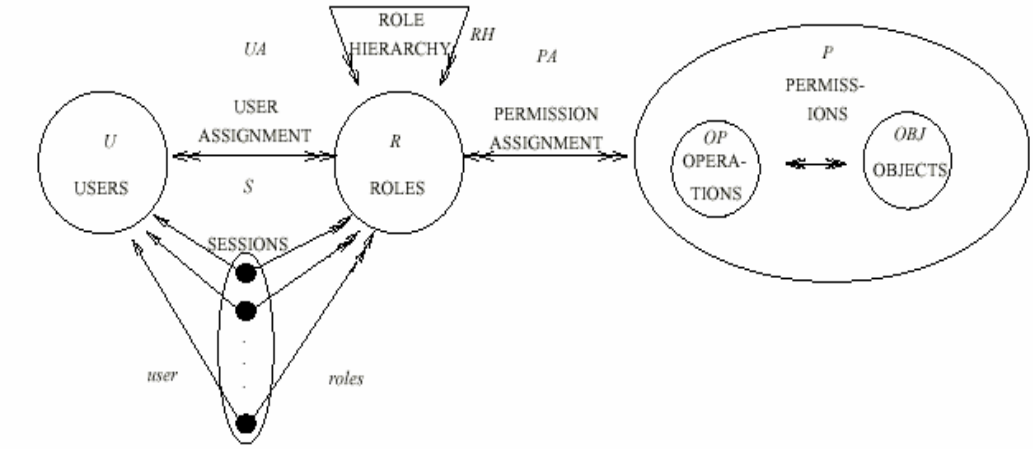


Figure 5. Entity-Relationship Diagram of RBAC-96 Schema Framework

5. VERIFICATION OF SCHEMA PROPERTIES AND IMPLEMENTATION CORRECTNESS

With the macro library of RBAC-96 definitions, we were able to demonstrate the two potential uses of RBAC schema analysis: (1) verifying the correctness of different RBAC implementation of an abstract access control model and (2) checking the consistency among entities, relations and constraints of a

RBAC schema as well as finding a plausible instance of the schema. We shall examine the first application in this section and study the second one in the next.

In order to verify the correctness of a RBAC implementation, ALLOY Constraint Analyzer must process the specification of two models: an abstract model that articulates the conceptual entities in an access control system, and a concrete model that describes the corresponding entities in a particular implementation. The Constraint Analyzer then tests the assertions of equivalent relations among corresponding entities of the two models: a positive thus affirms the correctness of the implementation while a negative result denies it.

In the project, we tested the correctness of a RBAC implementation of Bell-LaPadula confidentiality service against the abstract specification of its classic lattice model [Sect.3].

With RBAC-96 definitions, the roles and permissions that implement the Bell-LaPadula model can be specified by the following ALLOY code[5]:

```
state {
   // enumerating operations & roles
   partition ReadOp, WriteOp: fixed Operations!
   partition ReadRoles, WriteRoles: Roles
 }
def canReadRoles {
    all u | u.canReadRoles
        = { d | some r: u.UserRolesExt | d in ReadOp.RolePermissions[r] }
}
def canWriteRoles {
    all u | u.canWriteRoles
        = { d | some r: u.UserRolesExt | d in WriteOp.RolePermissions[r] }
}
```

After accepting the specification of the abstract model and its RBAC implementation, the ALLOY Constraint Analyzer was fed the following two assertions of equivalence between read/write permissions of the users in the abstract models and the corresponding roles in the RBAC implementation:

```
assert EquivCanRead {
   all u | u.canRead = u.canReadRoles
}
assert EquivCanWrite {
   all u | u.canWrite = u.canWriteRoles
}
```

The validity of the two assertions confirms the correctness of the RBAC implementation.

[5] Only the essential statements of the specification are included in this paper. Reader may refer to AVDA Project Report [Z+01] for the complete code listing.

6. VERIFICATION OF SCHEMA CONSISTENCY AND REALIZABILITY

In this section, we demonstrate ALLOY capability in verifying the consistency of a RBAC schema and searching for a plausible solution instance. We devised a simple schema of a simple four-person project as shown in Figure 6.

6.1 Roles

The RBAC model for the simple project consists of *four* roles :*Project Manager (ProjMgr), 1st Engineer (Engr1), 2nd Engineer (Engr2) and Tester*. These roles are linked by two relations:

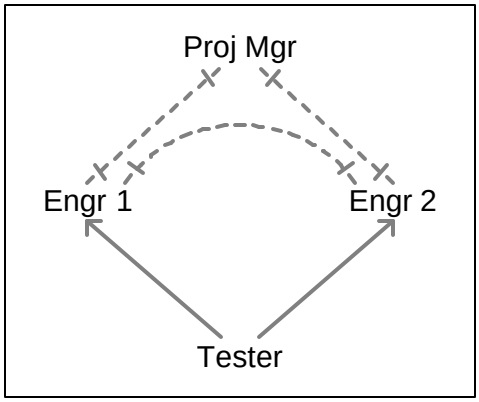


Figure 6.Project Role Hierarchy

Permission inheritance. Marked by arrows, the relation mandates that permissions are propagated along the direction of the arrows — the role at the end of an arrow inherits all the permissions of the role at the base of the arrow. In our model, both *Engr1* and *Engr2* inherit the permissions assigned to *Tester*.

Separation of Duty (SOD). Marked by *crossed dashes*, the relation mandates that the roles in the relation *cannot* be occupied by the same user. In our model, *ProjMgr, Engr1* and *Engr2* are linked by pair-wise SOD relations; thus, each of these roles must be occupied by a different user.

6.2 Read Permissions

The four roles are permitted to read from *three* types of files, *Management Files, Design Files,* and *Test Files*. Figure 7 shows the *read permissions* as arrows pointing at each role. Note that the inheritance relations among the roles are *consistent* with the read permissions; hence, no contradiction is detected and no other read permission can be inferred from inheritance.

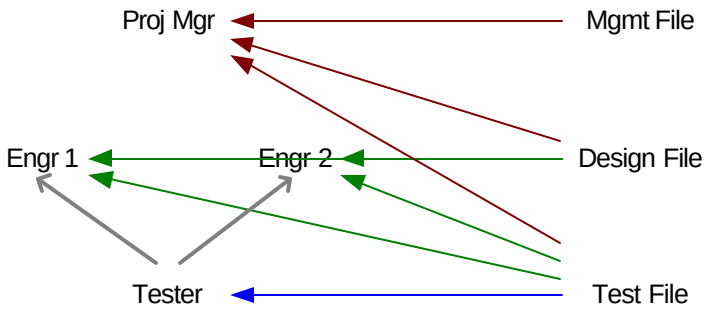


Figure 7. Read permission assignments for the small project team

6.3 Write Permissions

The four roles are also permitted to write to the three types of files. Figure 8 draws the *write permissions* using two different kinds of arrows: the *primary permissions* established explicitly by the administrators are drawn as solid arrows, and the *inferred permissions* derived from the inheritance relations are drawn as dashed arrows. The inferred permissions are imposed by the inheritance relations, which mandate the Engineers to have all the permissions assigned to the Tester.

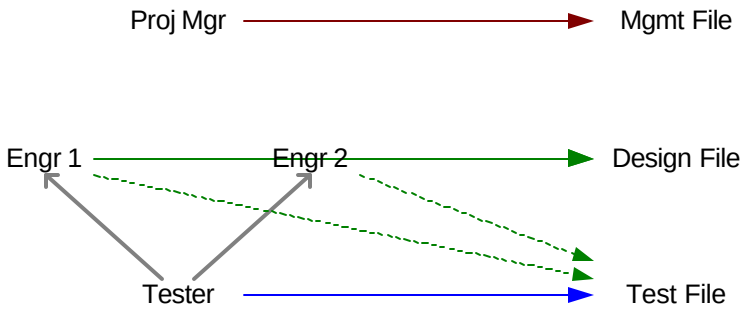


Figure 8. Write permission assignments for the small project team

6.4 User-Role Assignments

The Separation of Duty (SOD) relations among *ProjMgr*, *Engr1* and *Engr2* force these roles to be assigned to different individuals. Nonetheless, the Tester can be occupied by any of the people who are occupying the other roles since there is no rule prohibiting such arrangements. Figure 9 shows the minimum bindings between three people and four roles.

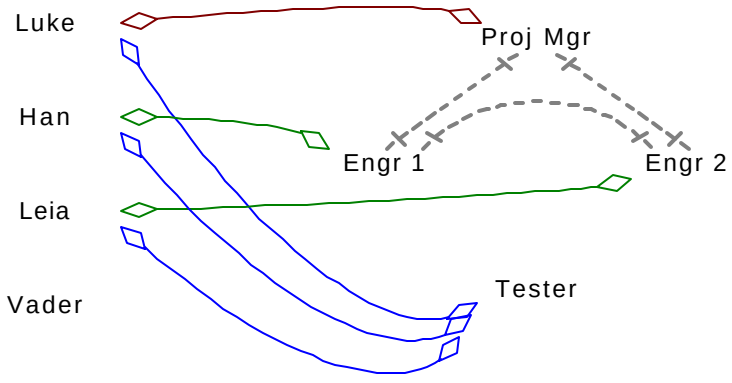


Figure 9. User-role assignments for the small project team

6.5 Full RBAC Model

We obtained the complete model of the project by combining all the relations mentioned in the previous sections. The entity-relation diagram of the entire model.

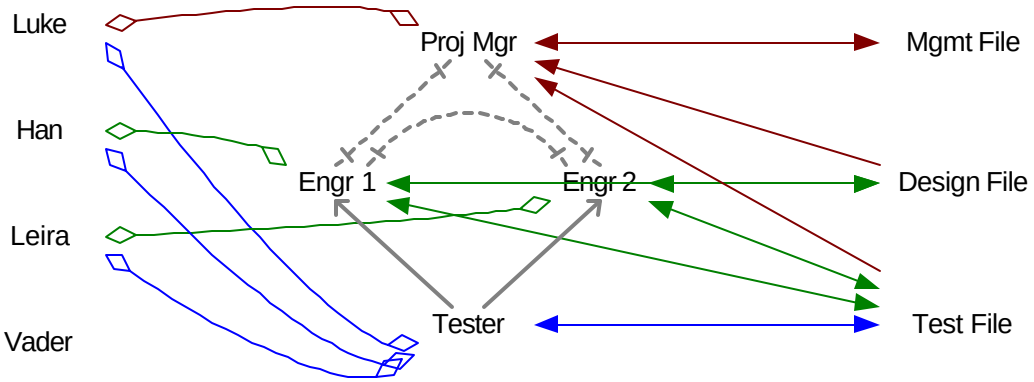


Figure 10. Full RBAC model for the small project team

7. CONCLUSIONS

From our exercises in using ALLOY in access control modeling and building an RBAC Schema Debugger, we can draw the following conclusions:

The ALLOY language has sufficient expressive power to prescribe implementation independent specification of access control systems. The examples we provided in this report provide sufficient evidence.

The ALLOY Constraint Analyzer can be used to verify the algebraic characteristics of access control schema. The confirmation of equivalence between the algebraic specification of the Bell-LaPadula model and its RBAC implementation demonstrated this capability.

The ALLOY Constraint Analyzer can be extended and used to verify the consistency (or to "debug") between entities and relations within an evolving RBAC scheme. The successful development of the RBAC Schema Debugger prototype confirms this ability.

8. REFERENCE

[AS00] Gail-Joon Ahn and Ravi S. Sandhu, "Role-based Authorization Constraint Specification." *Transactions on Information and System Security* 3(4) ACM Press, 2000.

[BLP75] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations and Models", M74-244, Mitre Corp., Bedford, MA, 1975. (Also available through National Technical Information Service, Springfield, VA, NTIS AD-771543).

[BK85] W. E. Boebert and R. Y. Kain. "A Practical Alternative to Hierarchical Integrity Policies." In *Proceedings of the Eighth National Computer Security Conference*, 1985.

[BN89] D. F. C. Brewer and M. J. Nash, "The Chinese Wall Security Policy", *Proceedings IEEE Symposium on Security and Privacy*, pp. 215–228, 1989.

[CZ01] J. Chu and J. K. Zao, "ALLOY-based RBAC Debugger Prototype – Test Case I: Simple Project Model," *AVDA Technical Notes*, BBN Technologies, July 2001.

[DW98] D. F. D'Souza and A. Cameron Wills. *Objects, Components and Frameworks With Uml: The Catalysis Approach*. Addison-Wesley, 1998.

[FK92] D. Ferraiolo, and R. Kuhn, "Role Based Access Control", *Proceedings of 15th National Computer Security Conference*, 1992

[G+96] J. Gu, P. W. Purdom, J. Franco and B. W. Wah, "Algorithms for the Satisfiability (SAT) Problem: A Survey", *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1996.

[GGF98] V. D. Gligor, S. I. Gavrila, and D. Ferraiolo. "On the formal denition of separation-of-duty policies and their composition". *Proceedings of IEEE Symposium on Research in Security and Privacy*, pp. 172–183, Oakland, CA, May 1998.

[GI96] L. Giuri and P. Iglio. "A formal model for role-based access control with constraints". *Proceedings of IEEE Computer Security Foundations Workshop*, pp. 136–145, Kenmare, Ireland, June 1996.

[Jac00a] D. Jackson. "ALLOY: A Lightweight Object Modeling Notation". Technical Report 797, MIT Lab for Computer Science , February 2000.

[Jac00b] D. Jackson. "Automating First-Order Relational Logic".*Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, San Diego, November 2000.

[Jac99] D. Jackson. "A Comparison of Object Modelling Notations: ALLOY, UML and Z". Unpublished manuscript. August 1999.

[JSS00] D. Jackson, I. Schechter and I. Shlyakhter. "Alcoa: the ALLOY Constraint Analyzer". *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[JSS01] D. Jackson, I. Shlyakhter and M. Sridharan, "A Micromodularity Mechanism", *Proceedings of ACM Foundations of Software Engineering / European Software Engineering Conference (FSE/ESEC '01)*, Vienna, Sept. 2001.

[LG00] B. Liskov and J. V. Guttag, *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*; Addison-Wesley, 2000

[OB85] Department of Defense, *Trusted Computer Security Evaluation Criteria*, DoD 5200.28-STD, 1985.

[PBSM00] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, S. Kent. "Domain Based Internet Security Policy Management". *Proceedings of DARPA Information Survivability Conference and Exposition (DISCEX'00)*, vol. 1, pp. 41–53, January 25–27, 2000.

[RJB99] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[San93] Ravi S. Sandhu, "Lattice-based Access Control Models." *IEEE Computer* 26(11): 9-19, IEEE Press, 1993

[SBM99] R. Sandhu, V. Bhamidipati, and Q. Munawer. "The ARBAC97 model for role-based administration of roles". *ACM Transactions on Information and Systems Security* 2(1):105–135, February 1999.

[SCFY96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E.Youman. "Role-based access control models". *IEEE Computer* 29(2):38–47, February 1996.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Second edition, Prentice Hall, 1992.

[SZ97] R. T. Simon and M. E. Zurko. "Separation of duty in role-based environments". *Proceedings of IEEE Computer Security Foundations Workshop*, pp. 183–194, Rockport, MA, December 1997.

[WBEM01] *Web-Based Enterprise Management Initiative*, Distributed Management Task Force (DMTF) Inc., July 2001, http://www.dmtf.org/standards/standard_wbem.php.

[WK99] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[XML01] *Extensible Markup Language (XML)*, The World Wide Web Consortium (W3C)., November 200], http://www.w3.org/XML/.

[Z+01] J. K. Zao, D. Jackson, H. Wee, J. Chu and R. Oliphant, "ALLOY-based Value and Dependency Abstraction (AVDA) Final Project Report", *BBN Technical Notes*, December 2001.

[Zha00] D. Zhang. *Collaborative Arrival Planner: Its Design and Analysis Using Object Modelling.* Masters of Engineering thesis. Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. May 2000.

Appendix A. ALLOY SPECIFICATION OF RBAC-96 FRAMEWORK

In order to specify general RBAC96 models (describable using RCL2000), we have written *domain*, *state* and *formulae* paragraphs that define the generic entities and relations used in RBAC96 and described in RCL2000. Following is the relevant excerpts.

```
model rbac96 {

    domain {Users, Roles, fixed Operations, Objects, Sessions}

    state {
        userRole : Users -> Roles                          //(UA): roles directly assign to user
        permissions : Operations -> Objects                //Permissions definition
        rolePermis [Roles]:  Operations -> Objects         //(PA): roles directly assigned to permissions
        userRoleExt: Users -> Roles                        //Roles* (u_i): all roles a user can fill
        objectOprToRole[Objects]: Operations -> Roles      //Roles (p_i) :
        objectOprToRoleExt [Objects]: Operations -> Roles  //Roles* (p_i): all roles that has permis
        roleObjToOperation[Roles]: Objects -> Operations   //Given Role & Objs, what operations poss?
        conflictRoles : Roles -> Roles                     //Role-centric SOD
        conflictObjects: Objects -> Objects                //Permissions-centric SOD
        conflictUsers: Users -> Users                      //User-centric SOD
        inherits : Roles -> Roles +                        //Set of roles that role inherits
    }

    // Inheritance -- Partial Order Relation
    //  Properties:
    //   1) Reflexive
    //   2) Transitive
    //   3) anti-symmetric
    inv inherits_qualities {
        all r | r in r.inherits
        all r | r.*inherits in r.inherits
        all r1, r2 | (r1 in r2.inherits && r2 in r1.inherits) -> r1 = r2
    }

    // Inheritance does not occur when 1 role's permissions is a subset of another
    //    For example, engr1 does not inherit from engr2
    //    Inheritance is a user-placed constraint
    inv inherits_inv {
        all r1, r2 | all opr |
            r1 in r2.inherits -> (all obj : opr.rolePermis[r1] | r2 in opr. objectOprToRoleExt[obj])
    }

    // userRoleExt:  set of roles, an individual role can takes
```

```
//   ie. all the roles >= this
def userRoleExt {
    all u : Users |
        u.userRoleExt = {r | some r1: u.userRole |  r in r1.inherits}
}

// objectOprToRole [Objects] : Operations -> Roles
//      Roles (p_i)   : all roles assigned to permission
def objectOprToRole {
    all o : Objects | all p : Operations |
        p.objectOprToRole[o] = {r | o in p.rolePermis[r]}
}

// objectOprToRoleExt [Objects] : Operations -> Roles
//    Roles*(p_i)  : all roles associated with permission (both assigned/inherited)
def objectOprToRoleExt {
    all o : Objects | all p : Operations |
        p.objectOprToRoleExt[o] = {r1 | some r : Roles | r in r1.inherits && o in p.rolePermis[r]}
}

// roleObjToOperation[Roles]: Objects -> Operations
//     Operations : given a role && obj, what operations are available
def roleObjToOperation {
    all r : Roles | all o : Objects |
        o.roleObjToOperation[r] = {p | o in p.rolePermis[r] }
}

//Rules for User-role conflicts
// - Enforces "inherited conflicts"
//    ex.  r1 conflicts r2  (no user can be both r1, r2)
//         r3 >= r2
//            -> r1 conflicts r3
// - non-reflexive
inv conflictRoleRule{
      no r  | r in r.conflictRoles  //irreflexive
      all u | no r1, r2, r3 |
            r1 in u.userRole &&
            r1 in r2.conflictRoles &&
            r2 in r3.inherits &&
            r3 in u.userRole
}

//All permissions in RolePermis must be an existing Permission
inv matchRolePermis {
    all r| all opr | all obj : opr.rolePermis[r] | obj in opr.permissions
```

```
        }

        //Define all permissions {
        inv definePermissions {
            all obj | all opr | obj in opr.permissions
        }
    }


//Constraints for Role-centric SOD
// -Irreflexive
// -Enforces rule: user cannot fill two conflictRoles (below rule)
// -Enforces "inherited conflicts"
//    ex.  r1 conflicts r2  (no user can be both r1, r2)
//         r3 >= r2 -> r1 conflicts r3
inv conflictRoleRule{
   no r | r in r.conflictRoles  //irreflexive
   all r1, r2 | r1 in r2.conflictRoles -> r2 in r1.conflictRoles  //symmetric
   all u | no r1, r2, r3 |
              r1 in u.userRole &&
              r1 in r2.conflictRoles &&
              r2 in r3.inherits &&
              r3 in u.userRole
}

//Constraints for User-centric SOD
// Irreflexive
// -Enforces rule: 2 conflict users cannot collectively fill 2 roles in conflict
//    (similar to conflictRoles)
// -  also conflictPermissions: 2 conflict users cannot collectively be capable of 2 permissions in conflict
inv conflictUserRule{
  no u | u in u.conflictUsers    //irreflexive
  all u1, u2 | u1 in u2.conflictUsers -> u2 in u1.conflictUsers //symmetric
  no u1, u2 |   u1 in u2.conflictUsers &&
                some r1, r2, r3 |
                   r1 in u1.userRole &&
                   r1 in r2.conflictRoles &&
                   r2 in r3.inherits &&
                   r3 in u2.userRole
}
```