ABSTRACT

James Anderson. ENFORCING ROLE-BASED ACCESS CONTROL IN A SOCIAL NETWORK. (Under the direction of Dr. Junhua Ding). Department of Computer Science, May 1, 2012.

Social networks supply a means by which people can communicate with each other while allowing for ease in initiating interaction and expressions. These systems of human collaboration may also be used to store and distribute information of a sensitive nature that must be secured against intrusions at all times. Given the massive operation embodied by social networks, multiple methods have been developed that control the flow of information so that those with authorization can gain access. Before allowing a social network to begin distributing its contents, a prudent prerequisite should be that the security protocols prevent unauthorized access.

Formal modeling and analysis of security properties, particularly those of Role-Based Access Control (RBAC), in social networks is the main focus of this thesis. A social network system and its security assurance mechanisms are modeled using the input language of Symbolic Model Verifier (SMV), and the properties of the system are specified using computation tree temporal logic (CTL*). Those properties are then verified using the SMV model checker. A real case was studied to demonstrate the effectiveness of model checking security properties in a social network system. The case consists of an account in which a group of users share various resources and access privileges which are controlled by RBAC. The case study results show that model checking is capable of formally analyzing security policies particularly RBAC in a social network system. In addition, the counter examples generated from model checking could help to

create test cases for testing system implementation, and they can help us to find defects in the model as well. Formally modeling and model checking security policies in a complex system, like a social network, can greatly improve the security of these systems.

ENFORCING ROLE-BASED ACCESS CONTROL ON A SOCIAL NETWORK

A Thesis

Presented to the Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

By

James Carold Anderson

May , 2012

ENFORCING ROLE-BASED ACCESS CONTROL ON A SOCIAL NETWORK

By

James Carold Anderson

APPROVED BY:

ACKNOWLEDGEMENTS

The completion of this thesis is a significant milestone in my life, not because of the degree it will earn me, but because of the personal and intellectual growth necessitated by the journey. In order to accomplish this I have had to set aside wasteful habits, abandon preconceptions and focus on achieving a distant and often frustrating goal.

First and foremost I would like to thank my thesis advisor, Dr. Junhua Ding. He dealt calmly with my obstinate procrastination and taught me to adopt a productive daily schedule which I will keep from now on.

No words can describe how indebted I am to my supportive family. To my father, who taught me to pursue my eclectic interests and encouraged a *civitas* mindset. I must also acknowledge my mother's constant sacrifice and effort to create a better life for her family, especially her children. Her love and dedication to her family motivated her to sacrifice everything for their good, ensuring opportunities denied herself and granting them a brighter future. I am thankful to my sisters, Tara and Heather, who imparted the wisdom they gained from their lives to me so that I could benefit from it as well.

Wassail eða hale við minn weddbroder! No matter the distance between us we shall forever be as close as the *Einherjar gebeorscipe njóta Valhöll*. The *gilpswide* and *beot* that I made over *meodugal* and *ydromellum* may always be *hatan hlíta*.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS OR ABBREVIATIONS

Chapter 1: Introduction

Web-based social network (WBSN) systems provide a communication median by which people can develop personal relationships through digital interactions. Each user creates an account that allows him or her to build a profile for creating and managing resources, such as blogs and picture albums, which can be viewed, altered, or commented upon by their fellow users in the system. These compilations of writings and graphics can be stored onto a social network account with the assurance that a specific category of user, such as those with similar backgrounds or cultural tastes, can locate and view them.

The urge to develop many positive relationships with others have led social networking users to connect with as many people as possible and not screen the motives of people requesting to link with their profile. (Hogben et al., 2007) Not all of the users are truly sincere in their desire for amicable relationships and are only interested in trying to exploit the information stored within social networks. Profiles and their contents have become targets of opportunity that can be used for malicious purposes.

One possible means by which a victim's profile can become a source of illicit profit is called digital dossier aggregation, which is the act of having a profile's contents downloaded and stored for use by third parties. (Hogben et al., 2007) This digital copy of the original profile can be a boon to those wishing to steal someone's identity or to commit fraud. Another way in which the contents of a profile can be used for threatening its owner are techniques such as Content-based Image Retrieval (CBIR), in which the features depicted in images can be used to determine the location where it was taken. (Hogben et al., 2007) CBIR can assist in allowing people, who wish to commit harm to a person, to discover the whereabouts of their targets.

Malicious users have also developed viruses and worms that can cause mayhem as soon as they are uploaded into a user profile. These programs can execute commands while disguising themselves as the user and perform inappropriate behavior, such as sending vulgar messages and/or images to coworkers and employers. Equally worse is when the rogue processes either steal or destroy the contents stored inside the profile.

For social networking sites to convince users to operate within their systems, they must convince the users that whatever is uploaded into their accounts can only be access by those that are authorized to and that their data and privacy is never intruded upon. To assuage these concerns various forms of security are implemented into these complex systems that will ensure that users will always be the sole controllers of the resources they upload.

1.1 Defcon Policy of Networking Systems

Complex information networks are not always used by a single organization, but are sometimes employed as a resource pool by multiple administrations. To better provide for their staff, networks can be contained in one core location that will allow for other sites to access it. A network can also be split into multiple partitions so that separate parts of an organization can access their allocated partition.(Migliavacc et al., 2010)The more domains that are incorporated into a system, each with their own methods of exchanging data, the harder it becomes to guarantee the integrity of the security policies. As the number of components used increases for each system, the chances that one component may have a bug that does not follow the security specification increases as well. Thus any, and all, information that is sent or within reach of this breach becomes accessible to unauthorized persons. This problem is further compounded by the fact that each system may be built by different developers, and that each could interpret the security policy incorrectly. Thus while the developers of one domain of the networks believe it is complying with the required security specification, it is actually countermanding it and allowing for intrusions.

The most desired arrangement of a system is the integration of different client entities that “enables the free flow of information and promises better service” (Migliavacc et al., 2010) and thus inspiring a process that is less interfering in the acquisition of resources. For software components of multi-domain systems to better align with each other, “they are often implemented as event-driven architectures, in which components, potentially belonging to different domains, must process and exchange data in the form of event messages.”(Migliavacc et al., 2010) Applications, such as those used by social networks, that share information between users with event messages must always follow the information handling policies. “This flow of sensitive data is an open problem of how to encode and enforce such flow-based security policies in the context of multi-domain event-driven applications.”(Migliavacc et al., 2010) This complication stems from the imbalance between the “high-level policies governing the handling of confidential data and low-level technical enforcement mechanisms.”(Migliavacc et al., 2010) “Traditional event-based and message-oriented software assists in correcting this malady by having the developers themselves enforcing the security policies inside of their applications.”(Migliavacc et al., 2010) This task is handled by the creation of policy enforcement points (PEPs) inside all of the components that will verify that the validity of sensitive operations before execution of any command. PEPs make use of access control policies to ensure that any data’s movement does not compromise the system’s security integrity. “However, these low-level mechanisms require the configuration of permissions at the lowest level of the system architecture. Each component would be required to outline all its requirements for approved data transfer and thus would be difficult to observe and police across a multi-domain environment."(Migliavacc et al., 2010)It is possible to alleviate some of the burden by implementing constraints on permitted data flows by the applications themselves. This method requires that "all domains must enforce the security policy uniformly across every domain by requiring all policy enforcement occur automatically when data flows across boundaries between components and domains and not perform tedious access control checks upon the whole scope of individual operations."(Migliavacc et al., 2010) An example of setting the system's allowed behavior is by implementing "an event-based middleware that enforces event flow security policy in distributed, multi-domain applications."(Migliavacc et al., 2010)The flow policy, called DEFCon Policy Language (DPL), sets the constraints on permitted component to component transactions through the use of security labels that outline the specifications of the security requirements. These labels will function as a set standard that all data flow must follow in order for components to send and receive the labeled data. This will force the system, no matter how many domains are incorporated at startup or added later, will adhere universally to the security specification.

1.2 Privacy Aware

WBSNs allow for participants to use the system in order to "share and publish information in the forms of annotations, blogs, etc., for a variety of purposes."(Carminati et al., 2008) The information to be shared with a person is determined by the relationship they have with others inside the network, which can be specified by assigning a trust level to each other. "The availability of this huge amount of information within a WBSN is a cause of concern for both the privacy of its users and the confidentiality of their information."(Carminati et al., 2008)To handle these concerns, WBSN began to implement safeguards in which the users can “decide whether their data, relationships, and resources should be public, accessible only by themselves, or by users with whom they have a direct relationship.”(Carminati et al., 2008)However, this form of access control is deemed too restrictive towards the goal of secure information sharing and too simple to ensure that unauthorized persons never can access their data. Desired model of “more flexible strategies, that allow a user to define his/her own rules, denoting the set of network participants authorized to access his/her resources and personal information.”(Carminati et al., 2008)

The Privacy-aware strategy is one form of security enforcement of WBSN resources where a user must present proof, of their existing relationship, to the resource’s owner before access is granted. This version of client-side access control requires that the relationships established by WBSN users are not revealed during operation. These relationships are further used in the WBSN’s security by first cataloging the relationships into types, depths, and trust levels. Afterwards, the relationships are “encoded through certificates and their protection requirements are expressed through a set of distribution rules, which basically state who can exploit a certificate for access control purposes.”(Carminati et al., 2008) The relationship is kept hidden by having the certificate encrypted with a symmetric key that is only sent to users listed in the distribution rules contained within the certificate. All certificates of the WBSN are stored and updated inside a secured central node which is trusted by all. Though more flexible than the earlier access control policy, there are a few weaknesses that can be exploited by malicious users. The creation of central node seems logical when dealing with a small number of users. However, WBSN are massive and can contain millions of users and would require an almost unmanageable set of certificates for all the possible connections and relationships that may exist inside of a system. “This bottleneck of performance makes the central node vulnerable to a Denial of Service attack.”(Carminati et al., 2008)

Another strategy, that does not use the central node to hold the certificates, allow for the WBSN to compensate for these shortcomings by having the users, in a collaborative effort, enforce security. Each of the owners’ resources would “regulate access only to users authorized by the owner. Also, the owner interacts only with resources of those that satisfy his/her own distribution rules.”(Carminati et al., 2008)Afterwards, any requester can be easily seen as either approved or not by seeing it is possible for them to access the resource. Therefore, every resource is aware of any relationships between all users and the distribution rules that must be followed. A user’s account will then only be invited to a collaboration of accounts if it satisfies all of the distribution rules of the whole group. Incorporating other forms of security, such as encryption and signature techniques, to reveal falsified certificates will further assist in enforcing the distribution rules.

While many of the methods described above can be used to enforce social network security specifications, they sometimes are not easily administered in reality. The varying degree of technical acumen required for a person to operate these techniques may make them logistically unfeasible in real life scenarios. An ideal form of verifying security protocol would be for a social network to use a method that does not require much technical ability of its supervisor while also being autonomous. This thesis seeks to demonstrate that a social network site’s access control specifications can be simulated and verified using model checking. Model checking would allow for a system administrator to confirm if the security properties, such as requiring that the system adheres to role-based access control (RBAC), are followed inside of a social network model and allow for the administrator to both locate and remedy any discovered errors.

1.3 RBAC and its implementation in WBSN

RBAC is frequently used as a median by which to control and restrict the actions of a system's users. Access control decisions are based upon a user's assigned role. This role commonly represents the position of the user in an organization's personal hierarchy and dictates what responsibilities the user will have. Each role implemented will have a pre-specified set of permissions that determine what objects the user with the role may have access to and what commands they will be able to execute upon the object. An additional benefit of RBAC is the ability to centrally control and maintain all of the existing access rights. The system administrators can determine which specific roles all of the users should have, and can easily alter them at a later time should the need arise.

While providing their communication services and applications, some WBSN are implemented with RBAC in order to prevent the unwanted disclosure of user information. Users are able to set and control their own access control policies. After the account holders have uploaded their materials, this process begins by the holders determining which roles of the system are allowed to interact with the stored resources. The level of interaction can range from just knowing their existence on the accounts profile to being able to view, write comments, or even add further materials upon it. Once the role-permission sets have been finalized, the owner of the account determines which users of the account should be given roles. Afterwards whenever that user accesses the account, they are given the pre-specified role and thus use their granted permissions upon the resources.

The RBAC policy of a WBSN give the users total control of all materials stored onto the network in a manner that does not require interference or even oversight from an administrative body. If implemented correctly, the administration can focus less on the security of the user to user interactions. However, should the policy arrange within the WBSN prove to be inadequate, the private materials of the networks users' that was only meant to be seen by a privileged few may be received by others. The possibility of such an occurrence requires no room for errors when setting the security specification. Therefore, WBSN must have an efficient means of thoroughly testing that the RBAC policies are followed inside of the network.

1.4 Model Checking

"Model Checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols."(Clarke et al., 1992) The specifications to be tested are written in a propositional logic that can express system changes over a period of time, and the reactive system itself is modeled in the form of a state-transition diagram to better display these changes. The process of determining if the representative model handles all of the specifications is done by searching through all possible states, and their transitions, then evaluating if they follow the desired model behavior.

One of the most advantageous features of model checking over other proof checkers is its ability for the procedure to be autonomous. After a detailed model representation of a reactive system and its specifications are created, the model checker will run to completion and terminate with a true answer if the model's behavior does not violate any of the outlined specifications. If, however, the model's behavior allows for a violation to occur, the model checker will output a trace of the model's states. This trace will show in what state of the model the fault was allowed to occur the thus shows why the specification was not satisfied. These counterexamples will allow for the user to determine which components of the system, and relevant specification, is the source of the failure.

Although a user can create models of great size and complexity to represent systems, it is unfeasible to try to represent many of the realistic systems due to their immense proportions. One method available to the users is to fabricate their models based on the modular structure of their desired system. In this way, the model can only consist of parts of the system that are vital to be correct in and during operation, and remove parts that have no need, or no requirement, to be represented in testing. “The specifications of the system can then be decomposed into properties that describe the behavior of small parts of the whole system.”(Clarke et al., 1992) The model checker can then singularly determine if each separate part of the system handles their local specification requirements. If all of sections return true, then by default the complete system will satisfy the complete set of specifications. Another way to represent systems of large size is to focus on the data paths. The symbolic model used to create the working model can be made to handle the system’s nontrivial data manipulations, but at the cost of making the verification process very complex. “This abstract approach is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system.”(Clarke et al., 1992) One such example is that one component in a circuit must output a value greater than the output of three specific components after some period of time. The abstraction curtails mapping all possible data values of the system’s parts in exchange of only working with a small set abstract data values. If collaborated with a states and transitions graph an abstract model of the system can be fabricated that would be both much smaller than the original model and easier to verify properties with.

Given the immense size of WBSNs and the complex security policies implemented within them, determining whether the system's RBAC properties are fully followed can be an onerous task. Model checking can fulfill this niche by creating a formal model that represents all parts of the network that must follow RBAC. The RBAC policies can then be then be added in the form of a meaningful logic specification that will be used to determine if the model is able to violate them. Model checking can determine whether the existing RBAC policy of the WBSN is adequate in ensuring that only authorized users can access the networks contents. Should a deficiency exist, the model checker will demonstrate how the trespassing user was able to interact with an object the system should have restricted from them. Counterexamples will report which system variables allowed the user to circumvent the failed RBAC policy and help determine how best to update the network and/or the access control policies for an effective security strategy.

1.5 Thesis Overview

This thesis explores the use of model checking in determining if a WBSN's RBAC policies are properly followed within the network. The process begins by building a formal model of the WBSN along with a set of specification representing its RBAC policies. The model checker will then analyze the system's behavior to determine if it is possible for the access control to be bypassed, and if so then determine which access control rule and component compromised the security enforcement.

In this work, a model of a custom social network is built to represent the proposed design of a company needed WBSN. This model represents not only the sharing of resources common in WBSN, but also the RBAC that is also implemented within it. The model is then sent as input to the model checker which verifies that all the RBAC properties are followed. In order to be sure that the model checker can find erroneous operations within the model, counterexamples are made in which each property are purposely made to fail. The model checker finds each of these problems within the design of the WBSN model that must be found early within the development of the system, or else the mistakes made in early development will problematic and costly later. Before the WBSN is deployed into usage, developers must be sure that its security follows the required RBAC properties. For if users are allowed access to a flawed network, a potentially devastating and unrecoverable leak of the sensitive and protected materials may occur.

Chapter 2 introduces the purpose and functionality of social networks. The security capabilities of complex systems will be presented along with multiple methods of access control.

Chapter 3 will cover the usage of model checking to determine system behavior. Also included will be the steps taken to create a model and the role that temporal logic has in assisting in the endeavor.

Chapter 4 outlines the model design that must be followed in order to properly emulate a social network profile. The access control policies governing an account's resources are covered in depth.

Chapter 5 covers the model checking procedure that is used to verify that the model created in chapter 4 adheres to the required access control polices of the account. The CTL specifications used by the model checker will be covered in depth.

Chapter 6 is the case study done on the subject of whether or not model checking can determine if a breach in the access control policies of the system can located. Experiences, both positive and negative, are covered in the process.

Chapter 7 concludes the work and provides an outlook of how the model checking process can be beneficial in developing proficient security protocols for a complex system.

Chapter 2: Social Networks

The proliferation of the internet has allowed people to rapidly gather information, and instantly communicate with each other. One of the most common communication methods is the use of social networks. A social network consists of a finite set of actors and the relations defined between them. (Wasserman et al., 1994)These relationships define the level of intimacy the actors have to each other in the social network as well as the level of interaction that is allowed between linked associates. Actors can range from being a single person to an entity consisting of multiple liaisons, such as a corporation. In a social network where the actors interact with as many people as they desire, the network's operation must ensure users that their uploaded data is secure against unwanted intruders. This process begins with the users' ability to confirm the identity of a user.

2.1 Identity Management System.

The Identity Management System is an important oversight feature of social networks. This process allows users to upload their data onto the social network and control how it is displayed and accessed. (Hogben et al., 1994) Users can decide what credentials others must have in order to view the data. For example, a user may determine that only those listed as a family member will be allowed to view all of their information. Another would be that members of a larger social group, like the employees of a company, will only allow coworkers access to the stored data. The user can even specifically target other users and control their access in great detail. Another feature of the Identity Management System is its ability to track who has accessed a user's data. This allows the user to make sure others are only querying the data they are supposed to and ensure that unauthorized access is not taking place. Should someone not follow the access control settings, the user can then take steps to restrict the trespasser's access to prevent future violations.

Once the customers of a social network are sufficiently assured that only the authorized users have access to their profiles, some may wish to expand the number of ways in which their profiles are accessed, such as with Application Program Interface(API). API allows users to take their social network profiles and frame it inside a third party web application. This allows increased data portability for the users but also raises concerns about maintaining security and privacy in such an easily accessible format. Strict authorization schemes and other forms of access control must be strictly enforced to protect data sent through this new median of connection. (Hogben et al., 1994)

2.2 Security of Social Networks

One implementation of social network security is a public/private key system. Upon joining a social network, the Identity Management System issues each of the users a token. These tokens contain the users' standard profile information as well as attribute levels of trust which cannot be alter. As the users interact with each other over the network the trust ratings of their tokens are altered. After many users vouch for the integrity of one user, that user's token will have a positive value. Should the users doubt the sincerity of a user, then that user's token will contain a negative value. If other users completely distrust one user, they may even revoke any association with the perpetrator using the system's certificate management and block the maleficent user from their perspective parts of the network. All of these token values assist users in determining if new acquaintances are trustworthy or insidious by evaluating the network's token opinion of them. In the event of a user wishing to move to another social network that uses the same token system, the user can have their token transferred with them to the new network.

This scheme is particularly useful in maintaining the appropriate level of privacy within a network. One could arrange it so that only people with a certain trust level and minimum level of interaction inside the network are able to view their information. The tokens, arranged in a private/public key setup, can even be used to send encrypted data between users (Hogben et al., 1994). However, this scheme relies upon the administration of a whole group of users to vouch for the integrity of others and leaves open the possibility of nepotism or corruption inside the system. A malicious person can create multiple accounts inside the network and have all the profiles rated positive. This will disguise their malignant nature by having an inflated positive trust value.

Though the public/private key system may be useful in certain informal collaboration networks, it cannot be relied upon to secure a user's account from intrusion since it relies too much on the input of other users. What is instead required is a form of security in which account holder can determine who else in the system can access their materials and to what degree. Thus many social networks include a form of access control in which users can manually set the access control requirements of their uploaded resources. The most frequently used are RBAC, Mandatory Access Control (MAC), and Discretionary Access Control (DAC).

2.3 Role-Based Access Control

RBAC is defined as allowing a system to clearly outline access control objectives in a mathematical and rigorous framework while also giving the users of the system a clear picture of the system's security arrangements and authoritarian obligations. (Sandu et al., 2001) RBAC has the actors of a social network interact with the objects that are contained within the network. These actors, however, are given specific roles to determine what actions they can execute, what objects they can access, and where in the system they can go. Roles do not always need to be directly given to a user. Instead, users can be encompassed into a group. This group is then assigned a set of permissions that all members of that group can use. (Sandu et al., 1994) Each of the objects created in the system have a level of clearance for all the data they contain. No actor can gain control of that object without a relevant role, and thus permission, assigned to them.

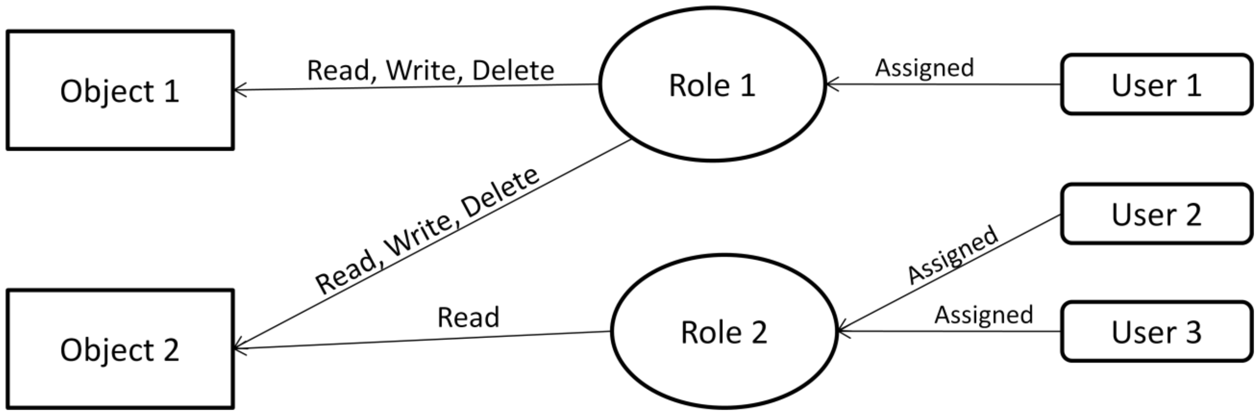


Figure 1: User/role relationships with objects

Roles define who inside of a system can access certain resources and how much interaction they can have with those resources. As seen in Figure 1, users of role 1 can execute read, write and delete on objects 1 and 2 while users of role 2 can only do read on object 2. An example of current systems that uses roles for operational purposes is Novell's Netware and Windows NT. An administrator of a server or a database benefit from implementing their systems with RBAC, which allows provides an easier format by which to police their domains. Some benefits of using them are: (Sandu et al., 1994)

- Access control decisions are based on “the roles that individual users take on as part of the organization”;
- Preferred in order to centrally control and maintain access rights that reflect the organization’s protection guidelines.

RBAC can also found in both operating systems and in user-level applications. Over time many variations of RBAC have been implemented. Some variations include whether or not relations exist between roles, roles and permissions, and users and roles. An example would be a role in which multiple users should not have access to, such as President of the United States. There should only ever be one person with this role at any one time, or else there could be an error. Some roles may even be able to inherit properties of other roles, so the system administrator does not need to repeatedly give permissions that are commonly used by everyone inside of the network. Roles can even be arranged so that a ranked hierarchy is made or that a separation of duties and responsibilities are outlined in the system.

One of the administrative benefits of roles is when system administrators must incorporate new users into their domain. It is far easier, and less technical, for a system administrator to give a new user a role with predefined permissions than to assign them the permissions directly. There is always the chance that the system’s permissions will be altered to meet a new security requirement, and so it would be less troublesome to only change the permissions to the finite set roles than it would be to change the permissions to a massive number of users individually.

To access the maximum benefit of RBAC system, three principles must be followed.

- Least Privilege: Only those permissions required for the tasks to be performed by the user are assigned to the role.
- Separation of Duties: Invocation of mutually exclusive roles can be required to complete a sensitive task, such as requiring both an accounting clerk and an account manager to participate in issuing a check.
- Data Abstraction: Instead of using permissions typical of operating systems, such as read and write, abstract permissions, such as credit and debit for an account object, can be implemented.

However, these principles are not fully necessary for every system, and the level to which each of the properties are followed is left to the system administrator to decide on their own. Take the separation of duties property, which can be separated as dynamic or static. In the static separation of duties, specific permissions should only be given exclusively to certain roles. Dynamic separation of duties requires analyzing the roles authorized to each user and requiring that users should not be given roles that do not conflict with the static separation of duties. An example of this is where a user, while in one role during a session, is not allow a specific permission, but may start again in a new session with a role that contains said conflicted permission.

Other forms of access control that are used in parallel with RBAC are that of discretionary access control, DAC, and mandatory access control, MAC. MAC is based upon the labeling of objects and users with security labels and only allowing interactions between those with similar labels. DAC is based on the users' permissions or denials to objects, whose access is arranged by the object's owner. While RBAC is a form of access control on its own, it can also be incorporated with elements of both DAC and MAC when the need arises.

2.4 Mandatory Access Control

MAC is a type of RBAC security commonly seen in database operations. Its main focus is the security of all accessed items while also ensuring secrecy. MAC begins this process by having all users and system objects to be assigned an attribute level. Afterwards, users in the system can only access an object with an equal attribute level. This is similar to the classification levels used on government documents, such that Top Secret documents are only known and accessed by the higher echelons of the intelligence community, and Unclassified materials are seen and given to almost anyone that requests it.

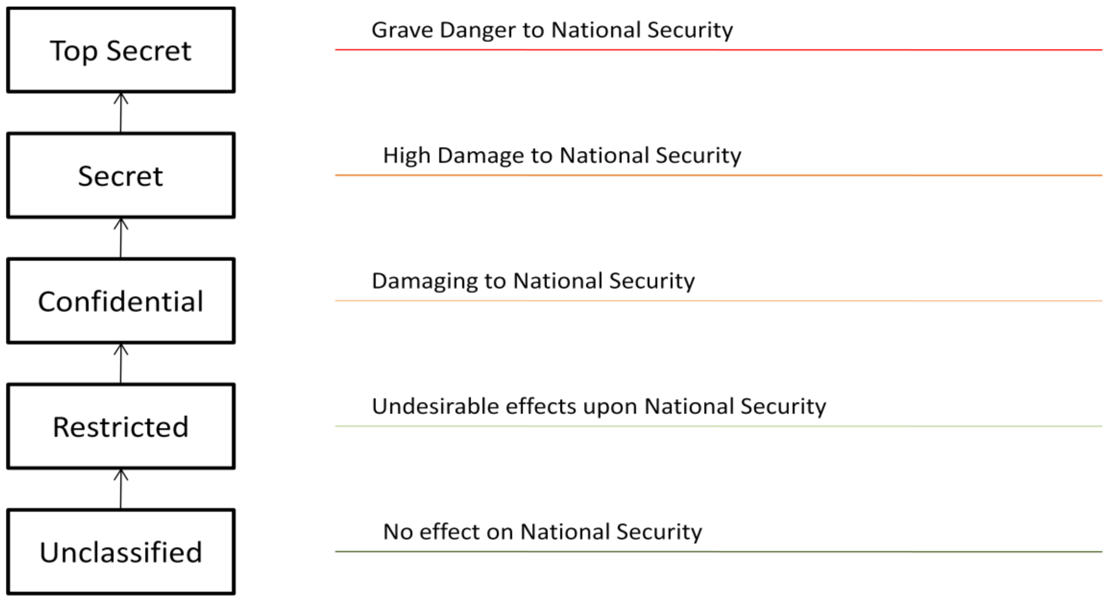

Top Secret
Grave Danger to National Security
Secret
High Damage to National Security
Confidential
Damaging to National Security
Restricted
Undesirable effects upon National Security
Unclassified
No effect on National Security

Figure 2: Classification Levels of Security in the U.S. Government

Some MAC environments take this step further by creating a multi-leveled security system. This is arranged by having the objects organized in order of their security level with “ability to access anything in the system is reviewed by a reference monitor.”(DoD et al., 1985) The reference monitor determines if the actor attempting to access an object has the required security clearance. Once approved the actor is allowed to perform actions that are allowed in his role. The reference monitor also ensures two rules are in effect. The first is the no-read-up rule, meaning that no user can access any object with clearance level higher than the one given in his/her role. The second rule is that of no-write-down, in which information flows from lower clearance levels to higher levels and prevents information from flowing from higher to lower levels. Thus a Top Secret document may contain references to a Classified document, but not the other way around. Having the flow of information fashioned as such helps to preserve secrecy. This is especially true in cases involving a Trojan horse virus inside of a MAC environment.

A Trojan horse is malicious code that is hidden within a program. The goal is for the system to treat the program as an actor with a role of the highest, or at least higher, clearance. This will allow the Trojan horse to use system authorized functions on highly restrictive objects, thus violating all security policy under the eyes of the reference monitor. Afterwards the Trojan horse will then undermine the information integrity by having system objects relabeled to a different clearance, or embed the data of a higher access object into a lower one. MAC principles prevent either from occurring by first having all data objects of the system predefined with no actor able to alter them. Second, MAC’s rule of no-write downs, in which data of higher clearance is never sent to a lower clearance document, prevents the Trojan horse of embedding sensitive data into an unrestricted file.

2.5 Discretionary Access Control

DAC is an access scheme in which the creator of an object determines who can interact with an object and what functions can be performed. (Osborn et al., 2000) The owner, who is usually the creator, is the only user that can set the permissions of the object. These permissions are then used to determine which commands specific users or group of users can perform upon the object. A DAC system in essence must follow three rules of operation (Osborn et al., 2000).

- The creator of an object is also the owner.
- The object will only have one owner.
- The deletion of the object can only be undertaken by the owner.

It should be noted that some systems allow for ownership of an object to change. These changes could be that the object is given to a new user, a user has taken a copy of someone else's object and makes it their own, or that ownership of an object is set by whoever uses the object last. Enforcement of this rule has led to variations into the DAC scheme.

1. Strict DAC is where the owner of the object is the only one who can set the permissions to the object. However whoever has read access to the object can easily copy its contents into their own object.
2. Liberal DAC is where the owner of the object can set the permissions and can even allow a set of users the ability of granting authority to other users. The number of repetitions of granting authority is delegated by the original owner of the object.

3. DAC With Changes to Ownership is where ownership of an object can be reissued and revoked by those accessing it. For example, one user may grant access to their object to another user. The new user can then grant access to whoever they want, and possibly causing a chain of grants. However, each user can still have their access permission revoked by the one gave it to them.

2.6 Forms of Authorizing Access Control Permissions

No matter which of the above forms of access control is implemented for a complex system, the main priority is to control the permissions given to each person. Information systems as we know them today offer services in which people can store, modify, and query the information that is contained within the system. (Thion et al., 2006) The main basis of the access control mechanisms that are used center around whether a subject, which could be a user or a process, is able to perform an operation, such as read or write, upon an object, such as a file or a folder. These operations upon objects are seen as system permissions. Permissions are usually not directly assigned to each user. As stated before, this would be time consuming and would lead to an increase chance of administrative mishandling. (Bertino et al., 2003) Instead, the permissions are categorized by the roles that need them within the system organization. The necessary roles are then given to the proper users and in turn, those users are only assigned permissions they need.

RBAC has three ways in which permissions can be implemented. Positive authorization is based on users having the required permissions to perform some action upon an object. For example, a user must have the write permission in order to execute write on an object. Systems using this for the basis of an access control policy are said to be following an open RBAC model.

Negative authorization is where the permissions deny users from interacting with objects. In this case, any user given the write permission cannot execute write upon the object. This access control policy is known as a closed RBAC model. There is also the option of using a combination of the two, which is known as hybrid RBAC, and also a form temporal RBAC that is based upon the amount of time in which a user has permission to an object. (Barker et al., 2003) If a system uses a hybrid model of RBAC, it is possible for a user to have positive authorization to an object and also have a conflicting negative authorization. In order to resolve this problem, "the system should support a conflict resolution strategy to determine which of the authorization policy should be followed."(Castano et al., 1995)

Chapter 3: Model Checking

Modeling is used to demonstrate the relative shape and purpose of an object. This is done either on something that was already made but hard to see in person or for an object before it is constructed. The latter is done to give people a general idea of what the actual article will appear when finished, the materials used to build it and to determine the estimation of the cost to construct the object. More thorough forms of modeling can even show points of weakness and assist in the removing of unnecessary or unwanted attributes from the design. One example would be a shipwright making a model in order to show the scaled design of a project. This will assist in planning the needed financial and material assets to begin construction.

3.1 Overview of Model Checking

In computers, modeling is used to simulate the design of hardware and software in order to determine the behavior of the system with a set of inputs. In the case of circuits, simulation is performed on a model design of a circuit which involve providing certain inputs and observing the corresponding outputs. (Clarke et al., 1999) This behavioral model simulation is then used with various scenarios to observe how the system operates, what types of errors can occur, and can the system handle an error exception without fatally crashing. This is very similar to Formal Methods, whose main purpose is to take the "applied mathematics for modeling and analyzing ICT systems."(Baier et al., 2008) While attempting to check the correctness with a mathematical mindset, Formal Methods can be made to work with both hardware and software designs. They are frequently used by multiple international organizations for their potential in detecting costly defects.

A model of a system's behavior is made to display the functionality and reactions during its runtime or show what flaws are currently present that may compromise its operation. This is called "deductive verification and involves the use of axioms and proof rules to prove correctness in the system." (Clarke et al., 1999) In the beginning this was arduously done by hand by developers to ensure that all possible test cases were taken into account. This time consuming process was overtaken with the use of software tools to allow for a systematic analysis of the proofs in the system.

Model Checking is a more preferable process to deductive verification of a concurrent finite state system. The overall process can be done without much manual input from the user and instead mostly involves the software checking all possible states automatically. "The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not and, if given sufficient resources, the procedure will always terminate with a yes/no answer." (Clarke et al., 1999)

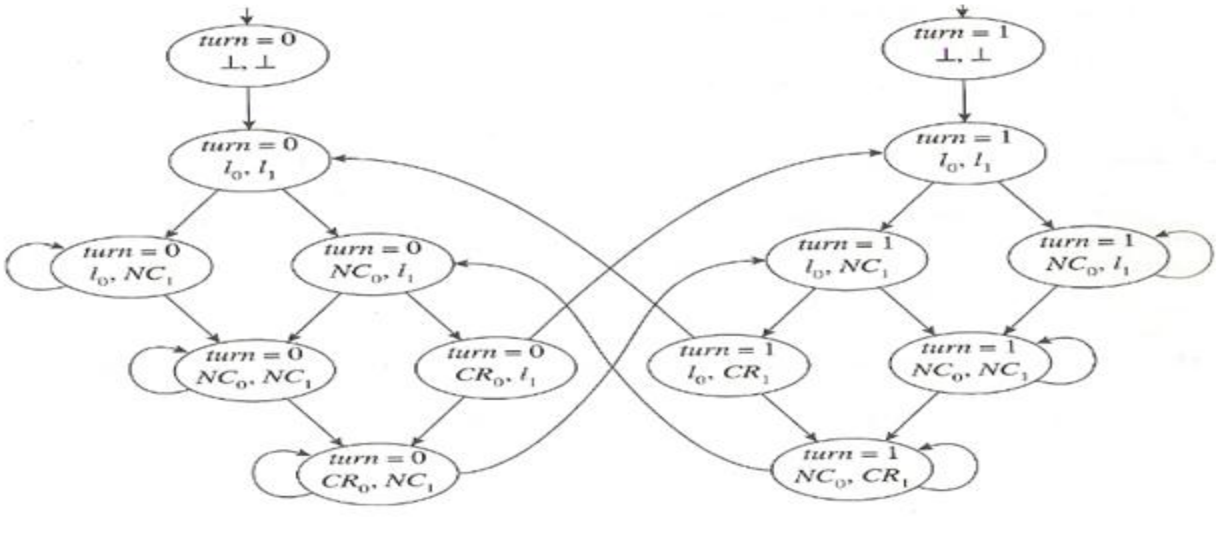


Figure 3: Kripke Structure for Mutual Exclusion (Clarke et al., 1999)

While the scope of the systems used may seem limited, due to the requirement of being finite, all hardware and software designs are finite state systems in reality and can be displayed in the form of a Kripke structure.“A Kripke structure is used to better illustrate the formal model and its various finite states by displaying the entire set of states, the transitions between states, and a function that labels each state with a set of properties that are true in this state.” (Clarke et al., 1999) The transitions displayed in a Kripke structure show the required actions needed to be taken by the system to reach the next state. This form of the model further assists the model checking process by demonstrating the cumulative after effects of the system’s operation over time.

Kripke structures are formally defined: (Clarke et al., 1999)

Let AP be a set of atomic propositions. A Kripke structure M over AP is a four tuple M = (S, S0, R, L) where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation that must be total, that is,

   $$\text{for every state } s \; \varepsilon \; S \text{ there is a state } s' \varepsilon \text{ such that } R(s, s').$$

4. $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

3.2 Steps of Model Checking

"Model-based verification is the process of creating a model of a system's behavior in a mathematically precise and singular manner." (Baier et al., 2008) Once accomplished, an accurate rendition of the system will usually allow the engineers to notice signs of incompleteness, ambiguities, and inconsistencies that would be expensive to repair at a later stage of the project's life cycle. "Model checking is in essence the exploration of all possible system states in a brute force manner."(Baier et al., 2008) Some problems that are examined in a system are classic computer science obstacles, such as deadlock and starvation. However, others can be more specific to the requirements of the project. Will the system always send a reply in twenty seconds, can the system consistently recover from failure during its lifetime of use, and will the system always be able to reach a certain state after an operation? Model checking can accommodate whatever requirements the developers must observe with the ability to track any necessary system states and further develop a realistic rendered model.

Once an accurate finite-state model of a system is created, the model checker will explore all relevant system states in an attempt to verify that the model follows all necessary properties. If the model checker should come across a state that shows otherwise, a counterexample will trace the path of how the system could reach such an unacceptable situation. A user will then be able to study the path to assist in remodeling the design. This entire process can be classified into the following three states: Modeling Phase, Running Phase, and Analysis Phase.

3.3Modeling Phase

The obvious first step to model checking is to begin building a model for the model checker to analyze. The system design to be considered is converted into the formalism accepted by the model checking tool. (Clarke, Grumberg, Peled 1999)Once the general shape is formulated, the system's model is then given an initial state." A state is a snapshot or instantaneous description of the system and captures the values of the variables at a particular instant of time." (Clarke et al., 1999) These states are used to show the transitional changes of the system during its operation and allows for the display of these transitions in the form of before and after shots.

After the proposed design of a system is finished, the choice of which properties will be verified and checked must be made. These properties can be anything from deadlock detection to unauthorized user access and can lead to an unlimited number of desired properties planned during the modeling phase, but sometimes only the most important are included into the model. This junction of the modeling phase is known as specification. Specification is handled by a form of logic formalism, called temporal logic, which is expressive of the system's behavior over time and has proved to be useful for specifying concurrent systems. (Clarke, Grumberg, Peled 1999) The syntax and semantics of the branch of temporal logic used in this thesis is discussed below.

3.3.1Temporal Logic

"Temporal logic is a formal logic for describing sequences of transitions between states in a reactive system while not mentioning time explicitly." (Clarke et al., 1999) This logic is also termed to be linear in nature since, as the Kripke structure is transversed over time, there is only ever one successor state that is used from the previous state. Temporal logic's specification formulas are designed to test if the model will enter a certain state eventually or never enter an error state.

Computation Tree Logic, or CTL, is a branching-time logic version of temporal logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the 'actual' path realized. (Huth et al., 2004) In order to selectively analyze the states of the model path quantifiers, which are used to express the branching of a computation tree, and temporal quantifiers, which are used to describe the properties of a path in a tree, are used. There are two path quantifiers used in CTL. **A** is used to represent "for all computation paths," and **E**, representing "from some computation path". **A's** meaning is that starting at a particular state, all of its successor states must uphold some property. **E** represents that starting at a particular state, there should exist a path to a successor state in which some property is upheld. Used in conjunction with these path quantifiers are five basic temporal operators:

- **X** ("next time") requires that a property holds in the second state of the path.
- **F** ("eventually" or "in the future") operator is used to assert that a property will hold at some state on the path.
- **G** ("always" or "globally") specifies that a property holds at every state on the path.
- **U** ("until") first property listed holds until a particular state in which the second property will then hold.
- **R** ("release") the second property holds along the path up to and including the first state where the first property holds, but does not require the first property to hold eventually.

3.3.2 Syntax of CTL

In reference to the aforementioned path quantifiers and temporal operators, CTL statements can be defined as such.

Definition (Huth et al., 2004)

CTL formulas are defined inductively via a Backus Naur form:

$$\phi ::= \bot \;|\mathrm{T}\;|\; p \;|(\neg\, \phi)|(\phi \;\vee \phi)|(\phi \wedge \phi)|(\phi \rightarrow\; \phi)$$

$$|AX\;\phi|EX\phi|AF\phi|EF\;\phi|AG\;\phi|EG\;\phi|A[\phi U\phi]|E[\phi U\phi]$$

3.3.3 Semantics of CTL

Definition: Let $M = (S, \rightarrow, L)$ be a model for CTL, $s \in S$, and ϕ a CTL formula. The relation $M, s \vDash\; \phi$ is defined by structural induction on θ:

1. $M, s \vDash\; T$ and $M, s\; \neg\!\vDash\!\bot$
2. $M, s \vDash\; p$ iff $p\; \in L(s)$
3. $M, s \vDash\; \neg\phi$ iff $M, s\neg\!\vDash \phi$
4. $M, s \vDash \phi_1 \wedge \phi_2$ if $M, s \vDash\; \phi_1$and $M, s \vDash\; \phi_2$
5. $M, s \vDash \phi_1 \vee \phi_2$ if $M, s \vDash\; \phi_1$or $M, s \vDash\; \phi_2$
6. $M, s \vDash \phi_1 \rightarrow \phi_2$ if $M, s\neg\!\vDash\; \phi_1$or $M, s \vDash\; \phi_2$
7. $M, s \vDash AX\;\phi$ iff for all s_1 such that $s\; \rightarrow\; s_1$ we have $M, s_1 \vDash \phi$. Thus, AX says: ‘in every next state.’

8. $M, s \vDash EX\ \phi$ iff for some s_1 such that $s \rightarrow s_1$ we have $M, s_1 \vDash \phi$. Thus, EX says: ‘in some next state.” E is a dual to A – in exactly the same way that ∃ is the dual to ∀ in predicate logic.
9. $M, s \vDash AG\ \phi$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1$equals s, and all s_i along the path, we have $M, s_i \vDash \phi$. Mnemonically: for all computation paths beginning in s the property ϕ holds Globally. Note that ‘along the path’ includes the path’s initial state.
10. $M, s \vDash EG\ \phi$ holds iff there is a path$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1$equals s, and for all s_i along the path, we have $M, s_i \vDash \phi$. Mnemonically: there exists a path beginning in s such that ϕ holds Globally along the path.
11. $M, s \vDash AF\ \phi$ holds iff for all paths$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1$equals s, there is some s_i such that $M, s_i \vDash \phi$. Mnemonically: for All computation paths beginning in s there will be some Future state where ϕ.
12. $M, s \vDash EF\ \phi$ holds iff for all paths$s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where s_1 equals s, there is somes_i such that $M, s_i \vDash \phi$. Mnemonically: for All computation paths beginning in s there will be some Future state whereϕ.
13. $M, s \vDash A[\phi_1\ U\ \phi_2]$ holds iff for all paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1$equals s, that path satisfies $\phi_1\ U\ \phi_2$, i.e., there is some s_i along the path, such that such that $M, s_i \vDash \phi_2$, and, for each $j < i$, we have $M, s_j \vDash \phi_1$. Mnemonically: All computation paths beginning in s satisfy that ϕ_1Until $\phi_2$holds on it.
14. $M, s \vDash E[\phi_1\ U\ \phi_2]$ holds iff there is a paths $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \cdots$, where $s_1$equals s, and that path satisfies $\phi_1\ U\ \phi_2$ as specified in 13. Mnemonically: there Exists a computation path beginning in s such that ϕ_1Until $\phi_2$holds on it.

3.4 Running and Analysis Phases

Once the specifications of the system properties have been finished, the running phase commences. This begins by giving the model checker as input the model M and a set of temporal logic specification ϕ. With S being the entire set of states in M, the model checker will transverse all states $s \in S$ to determine whether or not $s \vDash \theta$ holds. Once the model checker is finished, it will output all of the temporal logic specifications that were followed in side of the model. Should the model not uphold any of the formulas the model checker will print a history trace. This trace will show a path through the model's states to show how to reach the state where the error occurs.

An error can be discovered for many reasons during the model checking process. A likely occurrence is that of a memory error during the running phase, in which the model was too large for the program and thus had to abort. The only real solution, other than modifying the model checker to have more memory, is to break the model into multiple separate parts and test each one individually. The latter abstract process relies upon the fact that if all the parts of a system operate correctly and follow the specifications then the entire system should work as well. Another source of an error could be that the system was incorrectly modeled before any testing and analysis is attempted and thus returns vitiated results. This means that the model and its verification properties do not accurately reflect the required design of the system. A return to the modeling phase would be required to ensure that all faulty components and specification properties are removed or reconfigured. If the properties within the specification language are correct, then a flaw in the model is the cause of the error. The model has a state in which the system could compromise a specification property that must be upheld. The system design must then be improved to eliminate the flaw. However, if the specification properties are flawed and the model is actually correctly designed, then the analysis phase will report that an unacceptable state is reachable inside of the model. The only solution is to change the specification into a form that can properly test the given model. Once the verification phase of the model checking process is finished, the developers will be able to evaluate and correct the discovered of the deficiencies that exist inside of the current design of a system.

Chapter 4: Modeling RBAC in a Social Network

Before beginning to outline how the RBAC properties of a social network are to be represented by a formal model, the model checker must be discussed. In this paper, the New Symbolic Model Verifier (NuSMV) model-checking system is used to assist in determining if our social network model adheres to a set of temporal logic specification that will verify that the model upholds the required RBAC properties. NuSMV provides a language for describing the models and directly checks if the temporal logic formulas are valid inside of the model. (Huth et al., 2004) The model checker takes as input a text describing our social network model and print as output TRUE if a temporal logic specification holds within the model. If any of the specifications do not hold, the program will then print a trace showing why the specifications are false inside of the model.

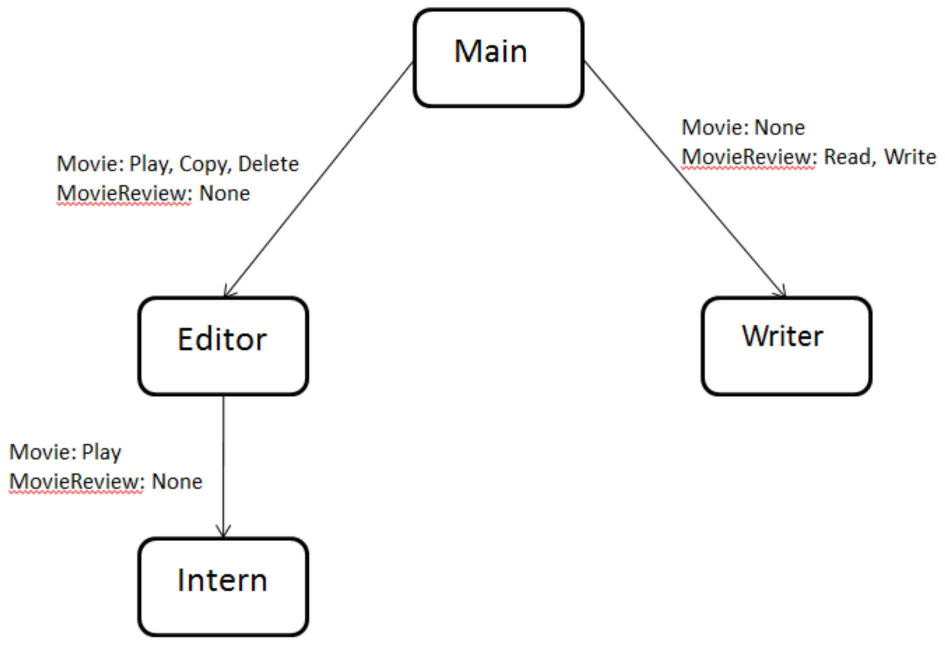


Figure 4: Model of the Example Network.

To use as an example, imagine a small social network consisting of the roles of editor, writer and intern. This network contains within it resources consisting of a movie and a review of the movie. Each role is granted specific permissions that allow them to execute commands upon the two resources. Writer's have the ability to Read and Write the movie review resource. Editors should be able to Play, Copy, and Delete the movie resources and Intern's should only be able use the Play command on movies. Editor's also have subordinate interns to whom the editor authorizes to use Play on the Movie Resource.

The SMV language of our model will consist of multiple modules which are identical to classes. Modules can declare variables and functions within their scope and reassign new values to variables during the operation of the model checker. To aid in readability and construction, our social network model will be broken down into various modules that are used to represent roles, user and resources. Just like in other programming languages, the program must start with a main module.

4.1 Overview of the Program Structure

```
MODULE main
```

The scope of the main module begins the line after this one and ends when another MODULE is written. This scope includes three sections. VAR, ASSIGN, and SPEC. The scope of main will include declarations of all of the resources needed for this model in the VAR section.

```
VAR

movie : videoResource();

movieReview : textFileResource();
```

In our example the needed resources are that of movie and the movie review. To model these resources a variable is declared with the names of movie and movieReview. The text to the right of the semicolon in the statement assigns a type to the variable. The videoResource() and textFileResource() are modules used to represent resources of a video and text type and will be covered later. By having the variables declared with their type specifer as the modules, each is a new instance of the module. Along with the resources, the permissions to each role connected to main must also be declared.

```
editorPermA : array 0..2 of boolean;

writerPermA : array 0..2 of boolean;
```

The editorPermA and writerPermA are variables of the permissions that roles editor and writer will have to the movie variable. The array following the semicolon tells us that the variables are an array and the 0..2 refers to the range of elements that will exist within the array, meaning zero to two. Each element represents a permission to the movie variable; zero for Play, one for Copy, and two for Delete. Following the of is the type of values the array will be allowed to contain, in this case Boolean. If the element of the permission is true, then the role will allow the user to have that permission to the movie resource. If the element is false, then the role will not allow the user to have that permission to the movie resource. Of course, since there are two resources the roles will need another set of permission arrays in this model.

```
editorPermB : array 0..1 of boolean;

writerPermB : array 0..1 of boolean;
```

The editorPermB and writerPermB follow the same principles of the previous variables except that they refer to the permissions the roles will have to the movieReview variable. The only permissions that the roles can send to movieReview are that of Read and Write. Thus the array is set to a range of zero to one with zero for the Read permission and one for the Write permission. Once the permission array variables are declared the values to their elements may then be given in the ASSIGN section.

```
ASSIGN

editorPermA[0] := TRUE;

editorPermA[1] := TRUE;

editorPermA[2] := TRUE;

editorPermB[0] := FALSE;

editorPermB[1] := FALSE;



writerPermA[0] := FALSE;

writerPermA[1] := FALSE;

writerPermA[2] := FALSE;

writerPermB[0] := TRUE;

writerPermB[1] := TRUE;
```

As explained within the example, the editor should only have the permissions to Play, Copy, and Delete the movie variable while not having Read and Write to the movieReview variable. The writer should only be assigned Read and Write to movieReview and should not have the Play, Copy and Delete permissions to the movie variable. To set the permissions each array's elements are assigned, represented by the := symbol, the necessary Boolean value to the role's required permission. Thus editorPermA's elements from zero to two are set to TRUE and editorPermB's element zero and one are set to FALSE while writerPermA elements zero to two are false and writerPermB elements zero to one are set to true. After the resource variables and the role permission arrays are set, main will then send those variables that are instances of the next role modules. This is accomplished by declaring two variables in VAR once again.

```
role1 : editorRole(movie, movieReview, editorPermA, editorPermB);

role2 : writerRole(movie, movieReview, writerPermA, writerPermB);
```

As with the movie and movieReview variables, role1 and role2 are instances of the editorRole and writerRole modules. This time however, the role modules require the instances of both resource objects and the related permission arrays as parameters. From these variables the model's development moves from main to the editorRole and writerRole modules, but before continuing on with the role modules, the resource modules must be explained.

4.2 Modeling the Resources

In main, the movie variable's type specifier is that of videoResource(), which makes the variable an instance of that module in the NuSMV file.

```
MODULE videoResource()
```

The module begins just like many using MODULE to state where the scope of videoResource() beings and what parameters it requires when a variable instance is created. By having the parentheses blank, this instance needs no parameters when declared. In our example, the commands that users can execute on movie are that of Play, Copy, and Delete. In order to model that an object of type videoResource() is receiving the commands a variable must be included that changes to the next user command the object receives.

```
VAR

state : {Wait, Play, Copy, Delete};
```

In the VAR section, a variable state is declared. Unlike the other variables so far, state is an enumerated type variable with the commands as possible values that state can be equated to. Wait is included in the enumeration for when a user does not send any of the commands or does not have permissions to execute any of the commands. Based upon this knowledge, a Kripke structure of video resource variable can be constructed. The Kripke transition values will be covered later on in the user module.

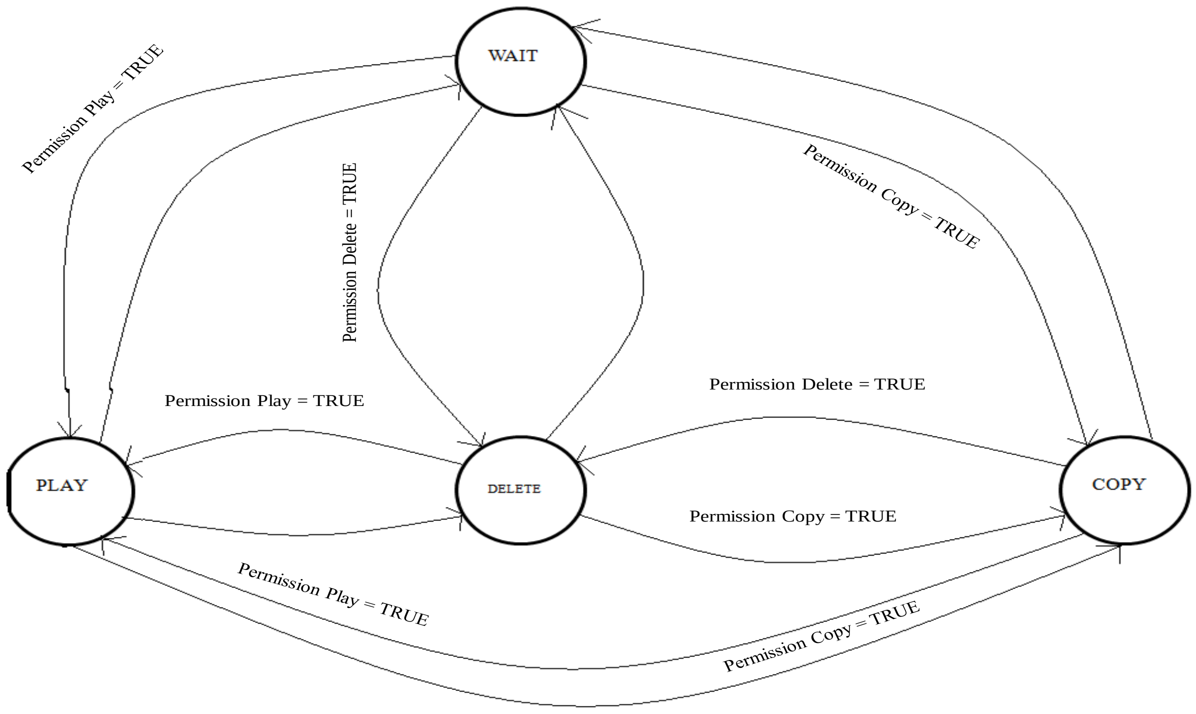


Figure 5: The Kripke Structure of a video resource with the requirements to reach each state.

Since state is an enumerated variable, its initial state must be assigned. Otherwise the model checker will set it at random.

ASSIGN

init(state) := Wait;

The init() function takes the variable within its parameters and sets its starting value to the enumeration following the := symbol. State is thus set to Wait since when the model begins, no one has sent any commands to the resource yet.

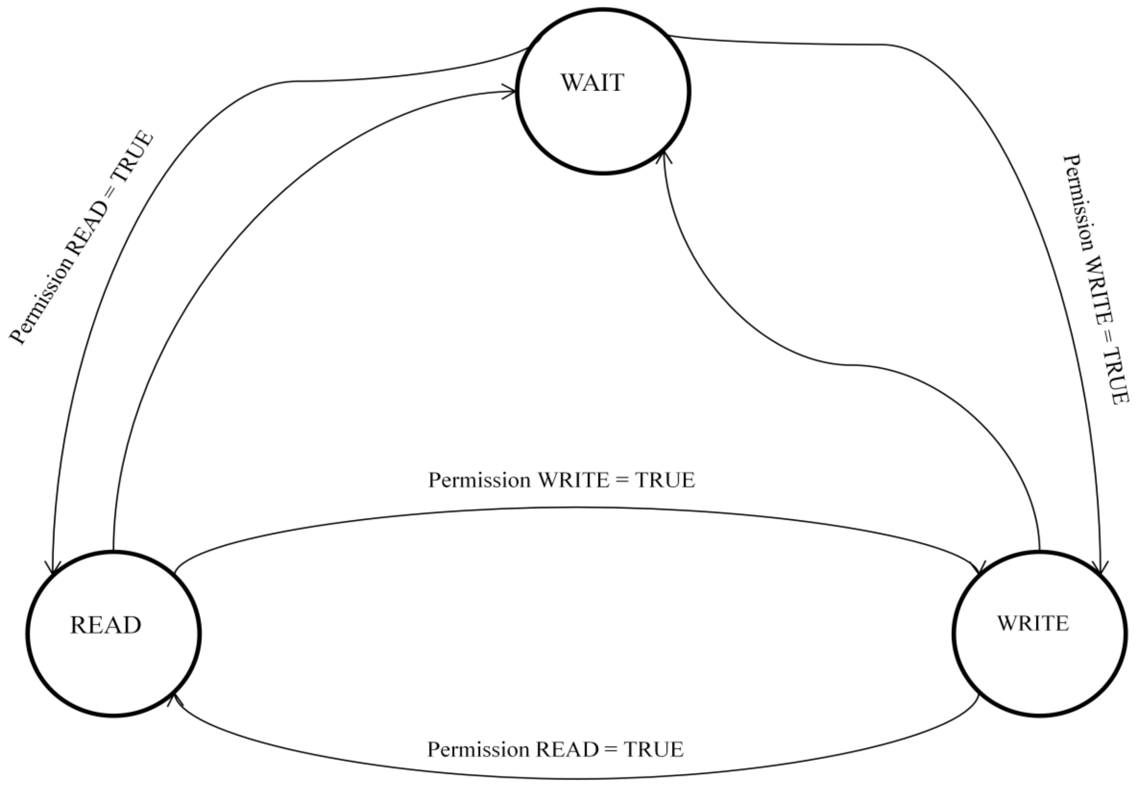


Figure 6: The Kripke Structure of a text resource with the requirements to reach the each state.

The textFileResource() module follows the same layout as videoResource().

```
MODULE textFileResource()

VAR

state : {Wait, Read, Write};

ASSIGN

init(state) := Wait;
```

The module has not passed any parameters from main. Its state variable is set to the possible commands users' can send the resource and the initial state is set to Wait.

4.3 Modeling the Roles

In main, the instances of variables of type editorRole and writerRole were made. These modules are passed instances of the resource variables created in main and the roles' permission arrays in order to model what permissions will be allowed to these roles inside of the network.

```
MODULE writerRole(movie, movieReview, writerPermA, writerPermB)

VAR

user2 : process User(movie, movieReview, writerPermA, writerPermB);
```

In the module writerRole, the system resources and the writer's permission arrays to those resources are received from main as parameters and passes them to the user2 variable, whose type specifier is a module titled User. The User module is used to model how users will be able to interact with the system resources based upon the permissions received from the role modules.

```
MODULE editorRole(movie, movieReview, editorPermA, editorPermB)

VAR

user1 : process User(movie, movieReview, editorPermA, editorPermB);

internPermA : array 0..2 of boolean;

internPermB : array 0..1 of boolean;
```

The editorRole module follows the same outline as the writerRole. It receives the resources and permission arrays from main and sends them to a variable that is an instance of the User module. However, unlike the writerRole, the editorRole module has a subordinate role underneath it and thus the function of the editor setting that role's permissions and passing them to their module must be modeled. This requires that a new set of permission arrays are created for the connected role module. As before in main, internPermA is a three element array of the permissions a user may have to the movie resource and internPermB is a two element array to the movieReview resource. These elements must then be set in the ASSIGN section with values required by our example.

```
ASSIGN

internPermA[0] := TRUE;

internPermA[1] := FALSE;

internPermA[2] := FALSE;

internPermB[0] := FALSE

internPermB[1] := FALSE;
```

As stated before, the intern role should only have the Play permission to the movie resource. Thus only interPermA[0] is set to TRUE and all others are made FALSE.

```
role3 : internRole(movie, movieReview, internPermA, internPermB);
```

The instances of the resources and the now assigned permission arrays are then passed as parameters to the role3 variable, which is an instance of the internRole module, in the VAR section.

```
MODULE internRole(movie, movieReview, internPermA, internPermB)

VAR

user3 : process User(movie, movieReview, internPermA, internPermB);
```

The internRole module takes the resource instances and permission arrays and uses them as parameters to its own User instance variable user3.

4.4 Modeling Users' Interaction with the Resources

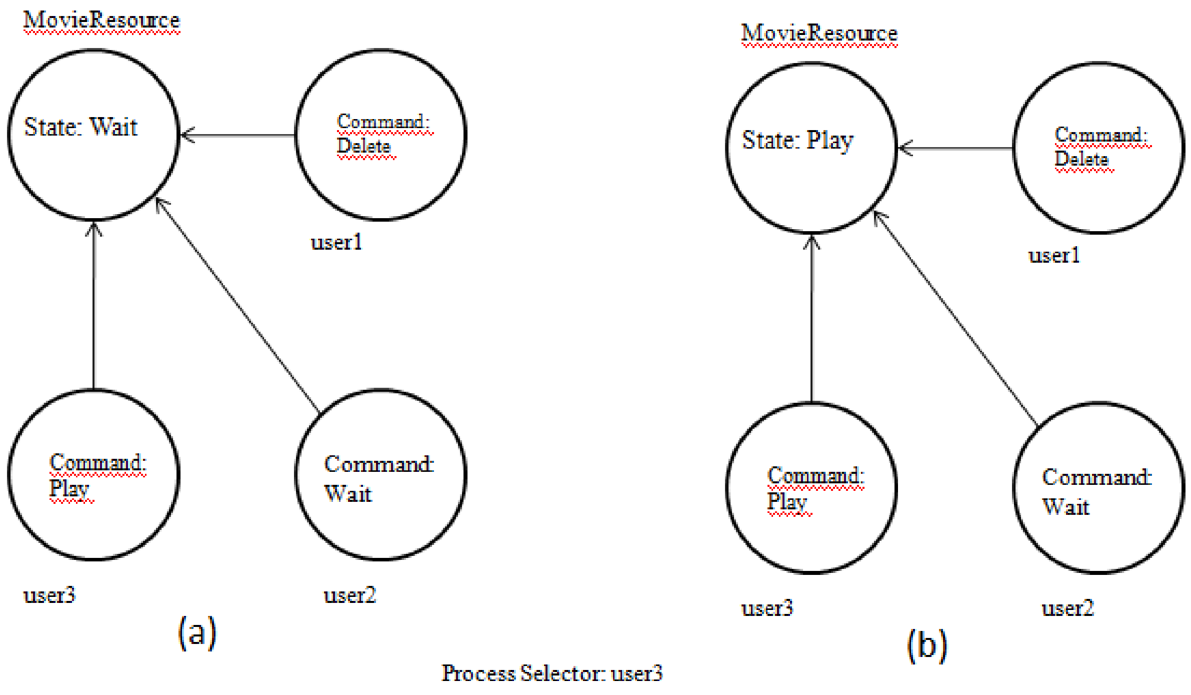


Figure 7(a): The movie resource is sent three different commands at the same during the running phase of the model checker. **(b)** The Process Selector of the model checker randomly chooses the user that the resource will accept.

The User module represents what commands a user will be allowed to send to the network resources. The user1, user2, and user3 variables are declared with a type specifier of the module User with their parameters being the resources and the permission arrays that were sent to the variables' role module. During the declaration of each variable, the keyword process is added after the semicolon. This is because every user variable will be interacting with the same instance of the resources and attempt to send commands simultaneously. Each resource, however, can only execute one command at a time. To ensure that only one command is accepted by the network resources, the user variables are designated as processes. During the running phase of the model checker, a variable called process_selecter will randomly choose one of the variables and have its command sent to the resources while the commands of the other user processes are ignored.

```
MODULE User(movie, movieReview, permA, permB)

VAR

myCommandA : {Wait, Play, Copy, Delete};

myCommandB : {Wait, Read, Write};
```

The User module shows once again that it is receives as parameters the same instances of the system resources and the permission arrays from the role module. Within the scope of the User module, the permission arrays passed to the module as parameters are called permA and permB. This is because it is unknown which role instance the User module was declared to. Two enumerated variables, myCommandA and myCommandB, are declared within the scope of the module and model the user sending commands to the resources. Variable myCommandA is the user's command to movie and myCommandB is to movieReview. Thus the possible values of the variables are identical to the state of the resources.

```
ASSIGN

init(myCommandA) := Wait;

init(myCommandB) := Wait;
```

Since the module has yet to analyze the permissions in the array, both myCommandA and myCommandB are set to the Wait value.

```
next(myCommandA) :=          case
```

The next() function takes the variable within its parentheses and sets its next value. The possible values of myCommandA are based upon the possible values of the user's permissions to the movie resource. To determine the next value a case expression is used, starting where the case keyword is displayed is used to analyze the values of the permission array.

```
  (permA[0] & !permA[1] & !permA[2]) : {Wait, Play};
(!permA[0] & permA[1] & !permA[2]) : {Wait, Copy};
  (permA[0] & permA[1] & !permA[2]) : {Wait, Play, Copy};
(!permA[0] & !permA[1] & permA[2]) : {Wait, Delete};
  (permA[0] & !permA[1] & permA[2]) : {Wait, Play, Delete};
(!permA[0] & permA[1] & permA[2]) : {Wait, Copy, Delete};
  (permA[0] & permA[1] & permA[2]) : {Wait, Play, Delete};
```

The statements following the case keyword are the cases of the case expression. To the left of the semicolon are the values the permissions must have in the case and the right contains the allowed next values of myCommand. Elements without the exclamation point must equal true and elements with the exclamation point must equal false. Examining the first case statement, the next value of myCommandA can be either Wait or Play should the permA[0] be true and permA[1] and permA[2] are false. However, if the user does not receive any permissions from their role a default case must be included.

```
TRUE   :  Wait;
```

```
                  esac;
```

The last case is exercised if none of the above cases are used and results in the user only sending Wait commands to the resource. The esac; is the key word used to end the switch statement.

```
next(myCommandB) := case
                    (permB[0] & !permB[1]) : {Wait, Read};
                  (!permB[0] & permB[1]) : {Wait, Write};
                    (permB[0] & permB[1]) : {Wait, Read, Write};
                    TRUE   :  Wait;
                  esac;
```

The function of next(myCommandB) follows the same outline as myCommandA except that there are only three possible cases, and the default, that may result from the second permission array. During the running phase of the model checker, the program will randomly select the value of the commands from the results of the relevant switch statement repeatedly. For example if the second case was used, the model checker will randomly select the values of Wait or Write for myCommandB and do so again and again while the model checker is operating. Whichever values are chosen for the user commands will be used to change the state of the resources.

```
next(movie.state) :=  case
                  myCommandA != Wait : myCommandA;
```

```
                        TRUE   : Wait;
                        esac;
next(movieReview.state) := case
                        myCommandB != Wait : myCommandB;
                        TRUE   : Wait;
                        esac;
```

The next function of the resources is used to change the values of the resources' state variable and thus models the users' ability to alter resources by the commands that they send. As before with the user commands, the next enumerated value of the states is determined by a switch statement of two cases. The first case being that should the user command to the resource not equal Wait, the value of the resource state is changed to the value of the command. Otherwise by default, the state is set to the Wait value.

4.5 Modeling User Sessions

In a complex system, users may be granted more than one role in order for them to perform their required tasks within the network. If the user is using a role that does not grant them the necessary permissions, they may log out and log back into the system with the appropriate role. In order to model this procedure the main module will need a set of variables containing what roles a user is assigned in the account.

```
  jamesRoles : array 0..2 of boolean;
  brianRoles : array 0..2 of boolean;
  jacobRoles : array 0..2 of boolean;

  willyRoles : array 0..2 of boolean;
```

In this example four users will be assigned roles that are implemented within the account. Those roles are intern, editor, and writer. The above four variables are arrays of three Boolean elements whose values will be used to determine the users' given roles. For example, should jamesRoles[0] be equal to true, then in our model the user James is able to log in as an intern. Element one refers to the editor role and element two is for the writer role. The elements of the arrays will have their values set in the ASSIGN section of the main module.

```
 jamesRoles[0] := TRUE;

 jamesRoles[1] := TRUE;

 jamesRoles[2] := FALSE;



 brianRoles[0] := FALSE;

 brianRoles[1] := FALSE;

 brianRoles[2] := TRUE;



 jacobRoles[0] := TRUE;

 jacobRoles[1] := FALSE;

 jacobRoles[2] := FALSE;
```

```
  willyRoles[0] := TRUE;

  willyRoles[1] := FALSE;

  willyRoles[2] := FALSE;
```

For our model, user James is to be allowed the roles of intern and editor. Brian will be assigned the writer role and users Jacob and Willy will only be permitted to log in as interns.

In the example model, only one user is allowed to use the editor and writer roles at any given time. To model the system preventing multiple users from using the same restrictive roles at the same time, a semaphore must be included.

```
   semEditor : semaphore();

   semWriter : semaphore();
```

In the VAR section of main, a variable is declared with a type specifier of semaphore for each of the two restrictive roles. These variables are thus instances of the semaphore module.

```
MODULE semaphore()

VAR

   sema : boolean;

   userName : { None, James, Brian, Jacob, Willy};
```

The semaphore module takes no parameters and declares a variables sema, of type Boolean, and an enumerated variable userName. Variable sema will be used to determine if a user will be allowed to log in as the role protected by the semaphore. If the value is true, the user is allowed to change their role to the exclusive role, and if it is false, the user may not. Variable userName's possible values are the names of the users that exist in the system and will be used to determine which of the users triggered variable sema.

```
ASSIGN

   init(sema) := FALSE;

   init(userName) := None;
```

Variable sema's initial value is set to false and userName is that of None.

```
VAR

   name1 : { None, James, Brian, Jacob, Willy};

   name2 : { None, James, Brian, Jacob, Willy};

   name3 : { None, James, Brian, Jacob, Willy};

   name4 : { None, James, Brian, Jacob, Willy};

ASSIGN

  name1 := James;
  name2 := Brian;

  name3 := Jacob;

  name4 := Willy;
```

Since the semaphores keep track of the user that is using the exclusive role, a variable must be initialized in main that will allow semaphore to know which user is specifically using the role. In the above VAR section of main, four enumerated variables are declared whose possible values are the names of the users in the account. They are each given an unused enumeration value in ASSIGN.

```
   userJames : process userSessions( semEditor, semWriter, jamesRoles, name1);

   userBrian : process userSessions( semEditor, semWriter, brianRoles, name2);

   userJacob : process userSessions( semEditor, semWriter, jacobRoles, name3);

   userWilly : process userSessions( semEditor, semWriter, willyRoles, name4);
```

As the last part of the process all of the semaphore variables, a paired set of role array and name variable are used as parameters for a set of specific named user variable instances of the userSessions modules. These variables model each user's ability to change roles between sessions in the account. The variables are also processes since multiple users may attempt to trigger the semaphores at the same time.

```
MODULE userSessions(semEditor, semWriter, givenRoles, myName)
```

```
VAR
  activeRole : {loggedOut, intern, editor, writer};
ASSIGN
  init(activeRole) := {loggedOut};
```

The userSessions module takes as parameters the semaphores declared in main, the name of the user represented in this instance and their assigned roles. The variable activeRole is declared in the scope of the module to track the status of the user's current role. ActiveRole is an enumerated variable with all user roles and loggedOut as a possible value. LoggedOut is used to represent when the user is no longer actively using a role and is not engaged in a session within the account. The initial value of activeRole is set to loggedOut since the module has yet to analyze the values in the givenRoles array.

```
  next(activeRole) := case
                        (givenRoles[2] & !semWriter.sema) : {loggedOut, writer};
                        (givenRoles[1] & !semEditor.sema) : {loggedOut, editor};
                        (givenRoles[0]) : {loggedOut, intern};
                        TRUE : loggedOut;
                        esac;
```

The next value of activeRole is determined by the results of a switch statement that examines the givenRoles array from highest to lowest with the relevant semaphore. If the element inside the array is true and the sema variable in the semaphore is false, activeRole's value is changed to that value. Should none of the switch statement cases be used, the result of the default case makes the activeRole variable set to loggedOut. For example, if user James only has givenRoles[1] set true and the semEditor sema variable if false, James's activeRole may become either loggedOut or the editor role as shown in the second case. If semEditor's sema is true, which means someone else is using the role at the time, the default case is evoked and user James's activeRole can only be set to loggedOut. Once semEditor's sema is equal to false again, the second case will result in user James activeRole being set to either loggedOut or editor.

```
  next(semEditor.userName) := case
                    activeRole = editor : myName;
                    TRUE : None;
                   esac;
  next(semWriter.userName) := case
                    activeRole = writer : myName;
                    TRUE : None;
                   esac;
```

Should the user's activeRole be one of the restricted roles, the semaphore of the role must be set to true in order to prevent another user from logging in with that same role. The above next functions change the value of the semaphore's userName variable to the name of user if that user's activeRole is equal to the role of the semaphore. It should be noted that these next functions of the semaphores' username variable are implemented after the sema variable next function since the value of userName must not be changed until after the sema is altered.

```
  next(semEditor.sema) := case
                        activeRole = editor : TRUE;
                        activeRole != editor & semEditor.userName = myName : FALSE;
                        TRUE : FALSE;
                      esac;
  next(semWriter.sema) := case
                        activeRole = writer : TRUE;
                        activeRole != writer & semEditor.userName = myName : FALSE;
                        TRUE : FALSE;
                      esac;
```

The first switch statement case of the semaphore results in the sema variable set to TRUE should the user's activeRole equal the semaphore role. The second statement is used when the user logs out of the exclusive role and sets the sema variable to false. When the user changes their activeRole out of the semaphore's exclusive role, their activeRole will not be equal to the role but the userName has not yet been changed. With the user no longer using the role and user's name stored as the person who activated the semaphore, the sema variable is thus set to false. If the first two cases are not used then by default the sema variable is false.

Chapter 5: Model Checking RBAC in a Social Network

During the modeling phase, a model is developed to represent the operation and features of a social network. Since the main priority and concern of a social network is the strength and efficacy of its security, the model and its checker will focus on the RBAC properties that must be correctly implemented by the Social Network.

The NuSMV model checker, works by taking a model as input and checks all existing and possible states of that model to determine if it fails to uphold a given set of specifications. These specifications, made in CTL and added to the model's modules, represents specific rules that the model must follow in order to adhere to RBAC. Should a discrepancy exist during the analysis phase of the model, the model checker will print a false value for the violated specification and traces the state transitions to where the trespass occurs. These statements provided by the model checker will not only assist developers in determining if their current access control implementation is operating as required, but also aid in locating why certain components are not following the specification.

5.1 Access Control Properties

In order to access any of the project resources existing within the social network account, users are given permissions from their role. These permissions determine what actions they are allowed to execute to the resources and thus ensures that a user is only able to perform commands dictated by their role. However, in order to ensure that any user is ever able to perform an action not licensed by their role, a CTL specification must be included for each of the permissions in the model. These specifications will check if a user is able to send a command to the resources while not having the required permission.

```
SPEC AG ! (permA[0] = FALSE & myCommandA = Play)
```

Above is the specification is added to the scope of the User module to test that access control for the Play command is followed. The keyword SPEC notifies the model checker that the line is a CTL statement and that should the model not adhere to the requirement of the temporal logic, a printout stating that the model is not compliant to the specification and a trace of why is made. The AG is the path and operation quantifier demands that the model checker examines all possible global states within the scope of the User module during the running phase. The rough translation of the specification is that in "all global states," myCommandA should not be equal to Play while permA[0] is equal to FALSE. In our model, permA[0] is the permission need for the user to send the Play command to the movie resource, and myCommandA is the variable that represents the user's current command to the movie resource. The possible values of myCommandA will be based upon a switch statement that analyzes all the elements of the permA array. The value of myCommandA should not be able to equal Play without permA[0] being TRUE and is the requirement of the specification for the model to pass.

```
                    TRUE   :  {Wait, Play};
```

To see if the model checker can discover a flaw in the model that violates the specification, the result of the default case for myCommandA is changed to allow the user to either use the Wait or Play command. Originally the default case is used when the permission array's elements are all false and thus the user is forced to only send the Wait command to the movie resource.

```
-- specification AG !(permA[0] = FALSE & myCommandA = Play) IN role1.userEditor is true
-- specification AG !(permA[0] = FALSE & myCommandA = Play) IN role1.role3.userIntern is
true

-- specification AG !(permA[0] = FALSE & myCommandA = Play) IN role2.userWriter is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  editorPermA[0] = TRUE

writerPermA[0] = FALSE

    role1.userEditor.myCommandA = Wait

  role1.internPermA[0] = TRUE

role1.role3.userIntern.myCommandA = Wait

role2.userWriter.myCommandA = Wait

-> Input: 1.2 <-

  _process_selector_ = role2.userWriter

  running = FALSE

  role2.userWriter.running = TRUE

  role1.role3.userIntern.running = FALSE
  role1.userEditor.running = FALSE

-> State: 1.2 <-

  role2.userWriter.myCommandA = Play
```

The output statement from the model checker reports that the access control specification for Play was followed by both the editor and intern users but not by the writer user. In our example model, the writer user's myCommandA switch statement results in the default case since the writer should not be able to send Play to the movie resource. The changes to the switch statement violate this principle and a trace is printed revealing the flaw. State1.1contains the starting values held by the variables within the model and shows that the permA[0] for editor and intern are both true while writer is false. Input1.2, the process selector, chooses a user process that will transition to a flawed state. The selector has the userWriter to run and shows that its myCommandA is changed to the value of Play even though its permA[0] is false.

```
SPEC AG ! (permA[1] = FALSE & myCommandA = Copy)
```

The access control specifications for the other permissions follow the same outline as the first. The above specification wants the model checker to verify that in "all global states" of the model, myCommandA should not be set to Copy if the user's permA[1], which is the permission element for Copy, is false.

```
                    (permA[0] & !permA[1] & !permA[2]) : {Wait, Play, Copy};
```

For a counterexample, the first case of myCommand is altered to allow for its value to be set to Copy while permA[1] is equal to false.

```
-- specification AG !(permA[1] = FALSE & myCommandA = Copy) IN role1.userEditor is true

-- specification AG !(permA[1] = FALSE & myCommandA = Copy) IN role1.role3.userIntern is
false

-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermA[1] = TRUE

  writerPermA[1] = FALSE

  role1.userEditor.myCommandA = Wait

  role1.internPermA[1] = FALSE

  role1.role3.userIntern.myCommandA = Wait

    role2.userWriter.myCommandA = Wait

-> Input: 1.2 <-

  _process_selector_ = role1.role3.userIntern

  running = FALSE

  role2.userWriter.running = FALSE

  role1.role3.userIntern.running = TRUE

  role1.userEditor.running = FALSE
```

```
-> State: 1.2 <-
  role1.role3.userIntern.myCommandA = Copy
-- specification AG !(permA[1] = FALSE & myCommandA = Copy) IN role2.userWriter is true
```

The model checker outputs that the userWriter and userEditor both followed the Copy specification and that userIntern did not. This is due to only the internRole user's myCommandA would have used the faulty case while the others did not. The value of userIntern's internPermA[1] is false in State 1.1 and that the user was still able change myCommandA into Copy.

```
SPEC AG ! (permA[2] = FALSE & myCommandA = Delete)
```

The Delete Specification wants the model checker to verify that in "all global states" of the model, myCommandA cannot be set to Delete the related permission element, permA[2], is false.

```
                    TRUE   :  {Wait, Delete};
```

The default case is changed to allow users without any permission to have myCommandA's value equal to either Wait or Delete.

```
-- specification AG !(permA[2] = FALSE & myCommandA = Delete) IN role1.userEditor is true
-- specification AG !(permA[2] = FALSE & myCommandA = Delete) IN role1.role3.userIntern is true
-- specification AG !(permA[2] = FALSE & myCommandA = Delete) IN role2.userWriter is false
```

```
-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermA[2] = TRUE

  writerPermA[2] = FALSE

  role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermA[2] = FALSE

  role1.role3.userIntern.myCommandA = Wait

  role2.userWriter.myCommandA = Wait

  -> Input: 1.2 <-

  _process_selector_ = role2.userWriter

  running = FALSE

  role2.userWriter.running = TRUE

  role1.role3.userIntern.running = FALSE

  role1.userEditor.running = FALSE

-> State: 1.2 <-
```

```
  role2.userWriter.myCommandA = Delete
```

The editor and intern users have both followed specification while the writer caused a failure. State 1.1 shows the writer not having true in permA[2], but has still executed Delete in its myCommandA in state 1.2.

```
SPEC AG ! (permB[0] = FALSE & myCommandB = Read)
```

The Read specification has the model checker verify that in "all global states" of the model, the user's myCommandB must not be equal to Read if permB[0], which is the permission element for Read, is false.

```
                    TRUE   :  {Wait, Read};
```

To test the specification, Read is included in the default case of myCommandB.

```
-- specification AG !(permB[0] = FALSE & myCommandB = Read) IN role1.userEditor is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermB[0] = FALSE
  writerPermB[0] = TRUE

    role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermB[0] = FALSE

  role1.role3.userIntern.myCommandB = Wait

  role2.userWriter.myCommandB = Wait

-> Input: 1.2 <-

  _process_selector_ = role1.userEditor

  running = FALSE

  role2.userWriter.running = FALSE

  role1.role3.userIntern.running = FALSE

  role1.userEditor.running = TRUE

-> State: 1.2 <-

  role1.userEditor.myCommandB = Read

-- specification AG !(permB[0] = FALSE & myCommandB = Read) IN role1.role3.userIntern is
false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample
Trace Type: Counterexample

-> State: 2.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermB[0] = FALSE

  writerPermB[0] = TRUE

  role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermB[0] = FALSE

  role1.role3.userIntern.myCommandB = Wait

  role2.userWriter.myCommandB = Wait

-> Input: 2.2 <-

  _process_selector_ = role1.role3.userIntern

  running = FALSE

  role2.userWriter.running = FALSE

  role1.role3.userIntern.running = TRUE

  role1.userEditor.running = FALSE
```

```
-> State: 2.2 <-

  role1.role3.userIntern.myCommandB = Read

-- specification AG !(permB[0] = FALSE & myCommandB = Read) IN role2.userWriter is true
```

The model checker reports that the editor and intern users did not follow the specification while the writer did. The first trace printed shows that the editor's permB[0] is false and that the user was still able to have myCommandB send Read to the resource. The second trace shows the intern user in the same circumstances.

```
SPEC AG ! (permB[1] = FALSE & myCommandB = Write)
```

The Write specification states that in "all global states" of the model, myCommandB should not be equal to Write if the Write permission in permB[1] is false.

```
                                TRUE   :  {Wait, Write};
```

Altering the myCommandB's default case to result in Wait and Write will test the specification.

```
-- specification AG !(permB[1] = FALSE & myCommandB = Write) IN role1.userEditor is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  movie.state = Wait
  movieReview.state = Wait

  editorPermB[1] = FALSE

  writerPermB[1] = TRUE

  role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermB[1] = FALSE

    role1.role3.userIntern.myCommandB = Wait

  role2.userWriter.myCommandB = Wait

-> Input: 1.2 <-

  _process_selector_ = role1.userEditor

  running = FALSE

  role2.userWriter.running = FALSE

  role1.role3.userIntern.running = FALSE

  role1.userEditor.running = TRUE

-> State: 1.2 <-

  role1.userEditor.myCommandB = Write

-- specification AG !(permB[1] = FALSE & myCommandB = Write) IN role1.role3.userIntern is
false
```

```
-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 2.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermB[1] = FALSE

  writerPermB[1] = TRUE

  role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermB[1] = FALSE

  role1.role3.userIntern.myCommandB = Wait

  role2.userWriter.myCommandB = Wait

-> Input: 2.2 <-

  _process_selector_ = role1.role3.userIntern

  running = FALSE

  role2.userWriter.running = FALSE
  role1.role3.userIntern.running = TRUE

  role1.userEditor.running = FALSE

-> State: 2.2 <-

  role1.role3.userIntern.myCommandB = Write

-- specification AG !(permB[1] = FALSE & myCommandB = Write) IN role2.userWriter is true
```

The model checker determines that editor and intern violates the specification while writer does not. The traces show that both the editor and intern users were not assigned the permission for Write, but were still able to have myCommandB equal to Write.

5.2 Permission Hierarchy

Permission Hierarchy is the property where if the permissions are arranged in a tier system, a user assigned a permission must also receive the permissions in the tiers below it. Taking the three permissions of the movie resource as an example, if the user was assigned the Copy permission they must also receive the Play permission as well.

```
SPEC AG (  (permA[1]) -> (permA[0]) )
```

The Permission Hierarchy specifications are added to the User module to monitor the values of the user permission arrays. The above CTL statement asks that in “all global states” should permA[1] be true implies, represented by the -> symbol, that permA[0] is true as well. Thus if the user is assigned the Copy permission, permA[1], they must also have been assigned the Play permission, permA[0]. As a counter example, the editorPermA[0]is set to false while editorPermA[1] is still true.

```
-- specification AG (permA[1] -> permA[0]) IN role1.userEditor is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermA[0] = FALSE

  editorPermA[1] = TRUE

  writerPermA[0] = FALSE

  writerPermA[1] = FALSE

  role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = FALSE

-- specification AG (permA[1] -> permA[0]) IN role1.role3.userIntern is true

-- specification AG (permA[1] -> permA[0]) IN role2.userWriter is true
```

The model checker reports the intern and writer users followed the specification since writer did not have any of the permissions to the movie resource while the intern only had its permA[0] set to true and not its permA[1]. The editor user did fail the specification as shown in the trace where in state 1.1 the editorPermA[0] is false and editorPermA[1] is true. This violates the specification property and must be fixed in order for editor to pass.

```
SPEC AG (  (permA[2]) -> (permA[0] & permA[1]) )
```

The second specification translates that in "all global states" of the model, permA[2] equal to true implies that permA[0] and permA[1] must also be true. Thus if the user was assigned the Delete permission, permA[2], they must also have been assigned the Play, permA[0], and Copy, permA[1], permissions as well. As a counterexample, the editor role will only be assigned the Delete and Play permissions to the movie resource.

```
-- specification AG (permA[2] -> (permA[0] & permA[1])) IN role1.userEditor is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermA[0] = TRUE
  editorPermA[1] = FALSE

  editorPermA[2] = TRUE

  writerPermA[0] = FALSE

  writerPermA[1] = FALSE

  writerPermA[2] = FALSE

  role1.userEditor.myCommandA = Wait

  role1.userEditor.myCommandB = Wait

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = FALSE

  role1.internPermA[2] = FALSE

  -- specification AG (permA[2] -> (permA[0] & permA[1])) IN role1.role3.userIntern is true

-- specification AG (permA[2] -> (permA[0] & permA[1])) IN role2.userWriter is true
```

The output of the model checker tells us that the writer and intern roles passed the specification since intern only had the Play permission and the writer user did not have any permissions at all for the movie resource. The reason for editor user specification failure is displayed in state 1.1 with editorPermA[1] equal to false in violation of the specification since the other elements of editorPermA are true.

```
SPEC AG (  (permB[1]) -> (permB[0]) )
```

The last specification is verifying that the permissions to the movieReview variable follow the role hierarchy property as well. The variables of type textFileResource receive the user commands of Read and Write. For the model to adhere to role hierarchy, if the user has the Write permission, then the user must also have the Read permission. The shown CTL statement wishes the model checker to verify that in “all global states”, if permB[1] is true implies that permB[0] is also true. Thus if the user has the Write permission, permB[1], then the user must also have been assigned the Read permission, permB[0]. As a counterexample, the writer user’s permB[0] is set to false in the main module.

```
-- specification AG (permB[1] -> permB[0]) IN role1.userEditor is true

-- specification AG (permB[1] -> permB[0]) IN role1.role3.userIntern is true

-- specification AG (permB[1] -> permB[0]) IN role2.userWriter is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermB[0] = FALSE

  editorPermB[1] = FALSE
```

```
writerPermB[0] = FALSE

writerPermB[1] = TRUE

role1.internPermB[0] = FALSE

role1.internPermB[1] = FALSE
```

In state 1.1, roles editor and intern are shown with having false in both elements of their permB arrays while writer has true in the higher element and false in the lower element of its permB array. This is in violation of the specification

For role hierarchy to be maintained within the account, users given a permission must also be assigned the lower permissions as well. Looking back at the switch statements of the myCommand variables in the User module, some of the cases used are in obvious violation of this property and should be removed. Should a user's permission arrangements equate to one of these faulty cases, they will then be redirected to the default case. This can lead to a new problem since the user was meant to have some permissions assigned to them from their role and are supposed to have some form of interaction with the objects. To prevent any complications, each role module is given a set of specification for a specific set of permissions their role must be assigned.

5.3 Minimum Duties

For a user to accomplish their tasks within a system, they must be assigned the needed permissions in order to do so. A CTL statement is added to each role module that verifies that the model has assigned the proper permission arrangements for the user roles.

```
SPEC AG( (internPermA[0] & !internPermA[1] & !internPermA[2]) &(!internPermB[0]
&!internPermB[1]))
```

The intern users in the example model are only supposed to be able to execute Play upon the movie resource and nothing more. The above specification, which is included in the internRole module, has the model checker verify that in "all global states" the element in internPermA[0] is true while internPermA[1], internPermA[2], and all of internPermB is false. As a counterexample, the intern role will be assigned the Read permission to the movieReview resource, internPermB[0], in the editorRole module.

```
-- specification AG (((internPermA[0] & !internPermA[1]) & !internPermA[2]) &
(!internPermB[0] & !internPermB[1])) IN role1.role3 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = FALSE

  role1.internPermA[2] = FALSE

  role1.internPermB[0] = TRUE

  role1.internPermB[1] = FALSE
```

The provided model checker printout shows that the minimum duties specification failed for the intern role module. State 1.1 shows that the intern incorrectly has the Read permission for the movieReview article.

```
SPEC AG( (editorPermA[2]) & (!editorPermB[0] & !editorPermB[1]) )
```

The minimum duties specification of the editor requires the permission of all the elements of permA and all the elements of permB to be false. The above CTL statement translates that in “all global states” of the model, the elements of editorPermA[2] is true and that element zero and one of editorPermB are false. The reason that only the second element of permA is required in the CTL statement is because permission hierarchy is also followed within the model and thus stating the lower two elements is rendered redundant. To test the specification, the editor role is only assigned Play and Copy in the main module.

```
-- specification AG (editorPermA[2] & (!editorPermB[0] & !editorPermB[1])) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  movie.state = Wait

  movieReview.state = Wait

  editorPermA[0] = TRUE
  editorPermA[1] = TRUE

  editorPermA[2] = FALSE

  editorPermB[0] = FALSE

  editorPermB[1] = FALSE
```

The printed statement, following the failure notification for the specification, shows that the editor role has only been assigned the lowest two permissions of the movie resource and thus users with an editor role cannot execute the needed commands to accomplish their required tasks.

SPEC AG((!writerPermA[0] & !writerPermA[1] & !writerPermA[2]) & (writerPermB[1]))

The writer role in the model is required for the permission of Read and Write to the movieReview resource. Thus the above specification is used to ensure that in "all global states" of the model, false is the value of all elements in writerPermA and true in writerPermB[1]. Since permission hierarchy must be followed in the model, only writerPermB[1] is needed in the CTL statement. As counterexample, the writer role is also assigned the Play permission, writerPermA[0].

```
-- specification AG (((!writerPermA[0] & !writerPermA[1]) & !writerPermA[2]) & writerPermB[1]) IN role2 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample
```

```
-> State: 1.1 <-

  writerPermA[0] = TRUE

  writerPermA[1] = FALSE

  writerPermA[2] = FALSE

  writerPermB[0] = TRUE

  writerPermB[1] = TRUE
```

The trace provided by the model checker show that the specification failure is caused by the writer role from having its first writerPermA element equal to true and must be to false for the model to uphold the minimum duties property of writer.

5.4 Static Separation of Duties (SSOD)

SSOD requires that the roles implemented within this system should only have permissions to one resource and not the other. The example model has the editor and intern only having permissions to the movie resource while the writer is only able to interact with the movieReview resource.

```
SPEC AG( (internPermA[0] -> AG !internPermB[0]) | (internPermB[0] ->  AG !internPermB[0])
)
```

The SSOD specification included in the internRole module has the model checker verify that in “all global states” internPermA[0] implies “all global states” of internPermB[0] are false or that internPermB[0] implies that “all global states” of internPermB[0] is false. This CTL statement equates to meaning that should the intern user’s lowest element in either permission array be true then the lowest element in the other array must be false. Since permission hierarchy is followed in the model, only the lowest elements need to be tested. For a counterexample, the intern's lowest elements to both permission arrays are set to true in the editorRole module.

```
        - specification AG ((internPermA[0] -> AG !internPermB[0]) | (internPermB[0] -> AG
!internPermA[0])) IN role1.role3 is false

        -- as demonstrated by the following execution sequence

        Trace Description: CTL Counterexample

        Trace Type: Counterexample

        -> State: 1.1 <-

          role1.internPermA[0] = TRUE

          role1.internPermA[1] = FALSE

          role1.internPermA[2] = FALSE

          role1.internPermB[0] = TRUE

          role1.internPermB[1] = FALSE
```

The trace shows that a true value exists in both arrays in violation of the specification and requires that one must be set false in order for the model to pass.

```
SPEC AG( (editorPermA[0] -> AG !editorPermB[0]) | (editorPermB[0] ->  AG !editorPermB[0])
)
```

```
SPEC AG( (writerPermA[0] -> AG !writerPermB[0]) & (writerPermB[0] ->  AG
!writerPermB[0]) )
```

The editor and writer roles have a near identical formula for their SSOD specification for their modules. The only difference is the variables to be used, the permission arrays to the roles, in the CTL statement. As a counterexample, both roles are given the lowest permission for an opposing resource, which is editor's editorPermB[0] and writer's writerPermA[0].

```
- specification AG ((editorPermA[0] -> AG !editorPermB[0]) | (editorPermB[0] -> AG
!editorPermB[0])) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  editorPermA[0] = TRUE

  editorPermA[1] = TRUE

  editorPermA[2] = TRUE

  editorPermB[0] = TRUE

  editorPermB[1] = FALSE

-- specification AG ((writerPermA[0] -> AG !writerPermB[0]) | (writerPermB[0] -> AG
!writerPermB[0])) IN role2 is false
```

```
-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 2.1 <-


  writerPermA[0] = TRUE

  writerPermA[1] = FALSE

  writerPermA[2] = FALSE

  writerPermB[0] = TRUE

  writerPermB[1] = TRUE
```

As expected, the roles' SSOD specification fails due to the role being given a permission to both resources. Thus for the model to adhere to SSOD, the roles must have a permission arrangement exclusive to only one of the resources.

5.5 Role Hierarchy

In order for the model to uphold the role hierarchy property, a superior role must have more permissions than its subordinate role. In this system, the only the editor has a subordinate role, the intern. Since permission hierarchy is implemented within the system as well, the editor must have a higher level permission than the intern for the model to pass. For example, if editor was assigned the Copy permission, the intern must not have an equal, Copy, or higher permission, Delete.

```
SPEC AG( (editorPermA[0] & !editorPermA[1]) -> AG !(internPermA[0] ) )
```

The first role hierarchy specification in the editorRole module requires that in “all global states”, editorPermA[0] being true and editorPermA[1] being false implies that in “all global states” it should not be possible for internPermA[0] to be true. The first half of the CTL statement is used to determine what the editor role’s permission level is. If the editorPermA[0] is true and the higher elements are not, then taking into account permission hierarchy editor must only have the Play permission. The second half of the specification is checking that the intern does not have the equivalent permission to editor. The reason that the higher permissions to intern’s arrays are not tested in the specification is due to permission hierarchy implemented within the system. The internPermA[0] can be true either because that permission was assigned to the intern role or because a higher permission was assigned. Either way will violate the specification. As a counterexample, the editor is given only the Play permission in main while the intern receives Play and Copy.

```
-- specification AG ((editorPermA[0] & !editorPermA[1]) -> AG !(internPermA[0])) IN role1 is
false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-
```

```
  movie.state = Wait

  movieReview.state = Wait

  editorPermA[0] = TRUE

  editorPermA[1] = FALSE

  editorPermA[2] = FALSE

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = TRUE

  role1.internPermA[2] = FALSE
```

The trace shows that the specification fails and the cause is due to a higher element of the intern role's permission array than the editor role's.

```
SPEC AG( (editorPermA[1] & !editorPermA[2]) -> AG !(internPermA[1] )  )
```

The second SSOD specification translates that in "all global states" of the model, editorPermA[1] is true and editorPermA[2] is false implies that in "all global states" internPermA[1] should be true. As a counter example, the intern and editor roles will both be assigned the Play and Copy permissions.

```
-- specification AG ((editorPermA[1] & !editorPermA[2]) -> AG !(internPermA[1])) IN role1 is
false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample
Trace Type: Counterexample

-> State: 1.1 <-

  editorPermA[0] = TRUE

  editorPermA[1] = TRUE

  editorPermA[2] = FALSE

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = TRUE

  role1.internPermA[2] = FALSE
```

The trace shows that the roles' permission arrays are equal and thus violate the role hierarchy specification requiring that editor role must have a higher permission setting than the intern role.

```
SPEC AG( (editorPermA[2]) -> AG !(internPermA[2]))
```

The last specification for the editor role permission to the movie resource checks that in "all global states" where editorPermA[2] is true implies that in "all global states" it should not be possible for internPermA[2] to be true. As a counterexample, both editor and intern's permA[2] is set to true.

```
-- specification AG (editorPermA[2] -> AG !internPermA[2]) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample
Trace Type: Counterexample

-> State: 1.1 <-

  editorPermA[0] = TRUE

  editorPermA[1] = TRUE

  editorPermA[2] = TRUE

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = TRUE

  role1.internPermA[2] = TRUE
```

The trace shows the equality of both the roles' permission arrays and thus violates the specification. The permission level of the subordinate role must be lowered in order for the model checker to approve of the model.

```
SPEC AG( (editorPermB[0] & !editorPermB[1]) -> AG !(internPermB[0]) )
```

The above specification requires the in "all global states" of the model, editorPermB[0] being true while editorPermB[1] is false implies that in "all global states" it should not be possible for internPermB[0] to be true. As a counterexample, editorPermB[0] and internPermB[0] are set to true.

```
-- specification AG ((editorPermB[0] & !editorPermB[1]) -> AG !internPermB[0]) IN role1 is
false

-- as demonstrated by the following execution sequence
```

```
Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  editorPermB[0] = TRUE

  editorPermB[1] = FALSE

  role1.internPermB[0] = TRUE

  role1.internPermB[1] = FALSE
```

The trace shows that the roles' permission arrays to the movie review resource are equal and violates the specification.

```
SPEC AG( (editorPermB[1]) -> AG !(internPermB[1]))
```

The specification translates that in "all global states" of the model, editorPermB[1] being true implies that in "all global states", internPermB[1] should not be true. As a counterexample, the editor and intern role are once again assigned true to element one of their permB arrays.

```
-- specification AG (editorPermB[1] -> AG !internPermB[1]) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-
```

```
  editorPermB[0] = TRUE

  editorPermB[1] = TRUE

  role1.internPermB[0] = TRUE

  role1.internPermB[1] = TRUE
```

As expected, assigning both roles the same permission to their permB arrays caused the specification to fail and requires that the subordinate role must have their permission array set to a lower level.

All of the role hierarchy specifications so far have checked that whatever the highest permission element to editor role's arrays is equal to true requires the intern role's highest element must be lower in order for the model to pass. However there is one case that must also be tested in the model. If the editor role does not have any elements equal to true in a permission array, the intern role must not have any true elements in that array as well. Thus if the editor role does not have Read, editorPermB[0], or Write, editorPermB[1], to the movieReview resource then intern role must not be assigned any permissions either.

```
SPEC AG( (!editorPermA[0]) -> AG !(internPermA[0]))
```

The specification above translates that in "all global states" of the model, editorPermA[0] equal to false implies that in "all global states" internPermA[0] must false as well. Thus if the editor role does not have the Play permission, editorPermA[0], then the intern role must not have the Play permission, internPermA[0], as well. As a counterexample, editorPermA[0] is set to false while internPermA[0] is true.

```
-- specification AG (!editorPermA[0] -> AG !internPermA[0]) IN role1 is false
-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  editorPermA[0] = FALSE

  editorPermA[1] = FALSE

  editorPermA[2] = FALSE

  role1.internPermA[0] = TRUE

  role1.internPermA[1] = FALSE

  role1.internPermA[2] = FALSE
```

The model checker reports that the specification has failed with the trace showing that the cause is because the intern role has the permission to Play for the movie resource, while the editor role does not have permission to perform any commands upon the same resource. In order for the model to pass the specification, either the editor role's must have a permission higher than the intern role or the intern role must not have any permission to the movie resource.

```
SPEC AG( (!editorPermB[0]) -> AG !(internPermB[0]))
```

Along with testing when the editor role does not have any permissions to the movie resource, a specification must also be included to test when the editor role does not have any permissions to the movieReview resource. The above specification translates that in "all global states” of the model, editorPermB[0] equal to false implies that in “all global states” internPermB[0] must also be false. Thus if the editor does not have the Read permission to movieReview, the intern does not have the permission either. As a counterexample, the internRole is assigned the Read permission to movieReview while the editor role has not been given the permission.

```
-- specification AG (!editorPermB[0] -> AG !internPermB[0]) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  editorPermB[0] = FALSE

  editorPermB[1] = FALSE

  role1.internPermB[0] = TRUE

  role1.internPermB[1] = FALSE
```

The model checker reports that the specification has failed and the trace shows that reason is because the editor role does not have any permission to the movieReview resource while its subordinate role, intern, has the Read permission. In order for the model to pass the specification, the editor role must have a higher permission than the intern role to movieReview or the intern role must not have any permissions just like the editor role.

5.6 Dynamic Separation of Duties (DSOD)

The SSOD's second property concentrates on a user receiving conflicting permissions from their roles during a session. DSOD, however, deliberates upon a user being assigned a pair or pairs of conflicting roles. This is because even though a user's current session role may not give that user any conflicting permissions, a user may later login with another role and possibly giving the user access to permissions that conflict with the last role. In order to have the model checker verify that the model does follow DSOD, a set of specifications must be added for each role that conflicts with another role. In the example model, only the editor and writer roles are in conflict with each other.

```
SPEC AG ( givenRoles[1] -> AG !(givenRoles[2])  )
```

The first DSOD specification, which is imbedded in the userRoles module, request that the model checker verify that in "all global states", givenRoles[1] being true implies that in "all global states" givenRoles[2] should not be true. The variable givenRoles is used to represent what roles the user has been assigned within the system. The element one is true when the user is granted the editor role, and element two is true when the user is granted the writer role. For the model to uphold the specification, the user must not be assigned the writer role while having the editor role. As a counterexample, userJames's roles have been altered to allow him all three roles of the system.

```
-- specification AG (givenRoles[1] -> AG !givenRoles[2]) IN userJames is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample
Trace Type: Counterexample

-> State: 1.1 <-

  jamesRoles[0] = TRUE

  jamesRoles[1] = TRUE

  jamesRoles[2] = TRUE

  brianRoles[0] = FALSE

  brianRoles[1] = FALSE

  brianRoles[2] = TRUE

  jacobRoles[0] = TRUE

  jacobRoles[1] = FALSE

  jacobRoles[2] = FALSE

  willyRoles[0] = TRUE

  willyRoles[1] = FALSE

  willyRoles[2] = FALSE

-- specification AG (givenRoles[1] -> AG !givenRoles[2]) IN userBrian is true

-- specification AG (givenRoles[1] -> AG !givenRoles[2]) IN userJacob is true

-- specification AG (givenRoles[1] -> AG !givenRoles[2]) IN userWilly is true
```

The model checker reports that all users, except userJames, passed the specification by not having the pair of conflicting roles of writer and editor assigned to them. In order for the model to pass, userJames must have one of the roles removed from his assigned set.

```
SPEC AG ( givenRoles[2] -> AG !(givenRoles[1])  )
```

The second specification included in the userRoles module has the model checker verify that when a user is granted the writer role, that they should not have the editor role. While this second CTL statement may seem unnecessary since the first specification would catch the same violation, it is included anyway to follow a best practice policy for systems with a larger set of user roles. If the system had a set of ten conflicting roles and a specification for each assists in determining the source of the model failure. If a user is assigned three out of the ten conflicting roles, the model checker will report that those three specifications have failed and reveal which roles must be removed from the user for the model to pass.

As a counterexample, userBrian is assigned the editor role while still having the writer role.

```
-- specification AG (givenRoles[2] -> AG !givenRoles[1]) IN userJames is true

-- specification AG (givenRoles[2] -> AG !givenRoles[1]) IN userBrian is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-
```

```
  jamesRoles[0] = TRUE

  jamesRoles[1] = TRUE

  jamesRoles[2] = FALSE

  brianRoles[0] = FALSE

  brianRoles[1] = TRUE

  brianRoles[2] = TRUE

  jacobRoles[0] = TRUE

  jacobRoles[1] = FALSE

  jacobRoles[2] = FALSE

  willyRoles[0] = TRUE

  willyRoles[1] = FALSE

  willyRoles[2] = FALSE

-- specification AG (givenRoles[2] -> AG !givenRoles[1]) IN userJacob is true

-- specification AG (givenRoles[2] -> AG !givenRoles[1]) IN userWilly is true
```

All the other users have passed the specification since they either only has the intern or editor roles and not the writer role. User Brian however causes the model to fail the specification since he has been granted both of the conflicting roles. In order for the model to pass the specification, user Brian must only be granted one of the conflicting roles.

While the above specification requires that no user is ever assigned a pair of conflicting roles, the example model also requires that multiple users must not be allowed to use role that should only be exclusively given to one person.

```
SPEC AG (  (  (userJames.activeRole = editor) -> AG  !((userBrian.activeRole = editor) |
(userJacob.activeRole = editor) | (userWilly.activeRole = editor) )))
```

The above specification asks to model checker to verify that in “all global states” of the model, userJames’s activeRole equal to editor implies that in “all global states” it should not be possible for userBrian, userJacob, or userWilly’s activeRole to be equal to editor. As a counter example userWilly will be assigned the same roles as userJames, both the intern and editor roles.

```
-- specification AG (userJames.activeRole = editor -> AG !((userBrian.activeRole = editor |
userJacob.activeRole = editor) | userWilly.activeRole = editor))  is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  jamesRoles[0] = TRUE

  jamesRoles[1] = TRUE

  jamesRoles[2] = FALSE

  brianRoles[0] = FALSE
  brianRoles[1] = TRUE
  brianRoles[2] = TRUE
  jacobRoles[0] = TRUE
  jacobRoles[1] = FALSE
  jacobRoles[2] = FALSE
  willyRoles[0] = TRUE
  willyRoles[1] = TRUE
  willyRoles[2] = FALSE
  userJames.activeRole = loggedOut
  userBrian.activeRole = loggedOut
  userJacob.activeRole = loggedOut
  userWilly.activeRole = loggedOut
-> Input: 1.2 <-
  _process_selector_ = userJames
  running = FALSE
  userWilly.running = FALSE
  userJacob.running = FALSE
  userBrian.running = FALSE

  userJames.running = TRUE

-> State: 1.2 <-

  userJames.activeRole = editor

-> Input: 1.3 <-

  _process_selector_ = userWilly

  userWilly.running = TRUE

  userJames.running = FALSE

-> State: 1.3 <-

  userWilly.activeRole = editor
```

The users Brian and Jacob upheld the specification as expected while users James and Willy cause the model to fail. State 1.1 shows that both roles are given the editor role by the variable givenRoles[1] being set to true. The process selector then chooses each conflicting user in order to show that user James has logged in as the editor in State1.2 and user Willy was also able to log in as the editor in State 1.3. User Willy must have not been assigned the editor role in order for the model checker to approve the model.

The specification for the exclusive role is only used to check when userJames is given the editor role and verify that no others have as well. However, if userWilly and userJacob were given the editor role and userJames was not, then the model checker will not catch the discrepancy. Thus each user must the same type of specification that checks to see if multiple users are able to log in with the exclusive role.

```
SPEC AG (  (  (userBrian.activeRole = editor) -> AG  !((userJames.activeRole = editor) |
(userJacob.activeRole = editor) | (userWilly.activeRole = editor) )))

SPEC AG (  (  (userJacob.activeRole = editor) -> AG  !((userJames.activeRole = editor) |
(userBrian.activeRole = editor) | (userWilly.activeRole = editor) )))

SPEC AG (  (  (userWilly.activeRole = editor) -> AG  !((userJames.activeRole = editor) |
(userBrian.activeRole = editor) | (userJacob.activeRole = editor) )))
```

As before, each specification asks the model to verify that in "all global states" where a user logs in with the editor role, no other user is also able to log in as editor. When all of these specifications are used in conjunction, we receive the same effect with the previous DSOD property showing exactly which users are in violation of the specification. To demonstrate, James and Jacob are both assigned the editor roles as a counter example.

```
-- specification AG (userJames.activeRole = editor -> AG !((userBrian.activeRole = editor |
userJacob.activeRole = editor) | userWilly.activeRole = editor))  is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  semEditor.sema = FALSE
```

semEditor.userName = None

semWriter.sema = FALSE

semWriter.userName = None

jamesRoles[0] = TRUE

jamesRoles[1] = TRUE

jamesRoles[2] = FALSE

brianRoles[0] = FALSE

brianRoles[1] = FALSE

brianRoles[2] = TRUE

jacobRoles[0] = TRUE

jacobRoles[1] = TRUE

jacobRoles[2] = FALSE

willyRoles[0] = TRUE

willyRoles[1] = FALSE

willyRoles[2] = FALSE

userJames.activeRole = loggedOut

userBrian.activeRole = loggedOut

```
  userJacob.activeRole = loggedOut

  userWilly.activeRole = loggedOut

-> Input: 1.2 <-

  _process_selector_ = userJames

  running = FALSE

  userWilly.running = FALSE

  userJacob.running = FALSE

  userBrian.running = FALSE

  userJames.running = TRUE

-> State: 1.2 <-

  userJames.activeRole = editor

-> Input: 1.3 <-

  _process_selector_ = userJacob

  userJacob.running = TRUE

  userJames.running = FALSE

-> State: 1.3 <-

  userJacob.activeRole = editor
```

```
-- specification AG (userBrian.activeRole = editor -> AG !((userJames.activeRole = editor |
userJacob.activeRole = editor) | userWilly.activeRole = editor))  is true

-- specification AG (userJacob.activeRole = editor -> AG !((userJames.activeRole = editor |
userBrian.activeRole = editor) | userWilly.activeRole = editor))  is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 2.1 <-

  semEditor.sema = FALSE

  semEditor.userName = None

  semWriter.sema = FALSE

  semWriter.userName = None

  jamesRoles[0] = TRUE

  jamesRoles[1] = TRUE

  jamesRoles[2] = FALSE

  brianRoles[0] = FALSE

  brianRoles[1] = FALSE

  brianRoles[2] = TRUE
  jacobRoles[0] = TRUE

  jacobRoles[1] = TRUE

  jacobRoles[2] = FALSE

  willyRoles[0] = TRUE

  willyRoles[1] = FALSE

  willyRoles[2] = FALSE

  userJames.activeRole = loggedOut

  userBrian.activeRole = loggedOut

  userJacob.activeRole = loggedOut

  userWilly.activeRole = loggedOut

-> Input: 2.2 <-

  _process_selector_ = userJacob

  running = FALSE

  userWilly.running = FALSE

  userJacob.running = TRUE

  userBrian.running = FALSE

  userJames.running = FALSE
```

```
-> State: 2.2 <-

  userJacob.activeRole = editor

-> Input: 2.3 <-

  _process_selector_ = userJames

  userJacob.running = FALSE

  userJames.running = TRUE

-> State: 2.3 <-

  userJames.activeRole = editor

-- specification AG (userWilly.activeRole = editor -> AG !((userJames.activeRole = editor |
userBrian.activeRole = editor) | userJacob.activeRole = editor))  is true
```

Only two of the four specifications failed within the model. The two that pass, Brian and Willy, shows that those two users have nothing to do with the exclusive role and can be ruled out during the debugging process. Only users James and Jacob need to be examined, which the trace shows both being able to log in as the editor and violate the specification.

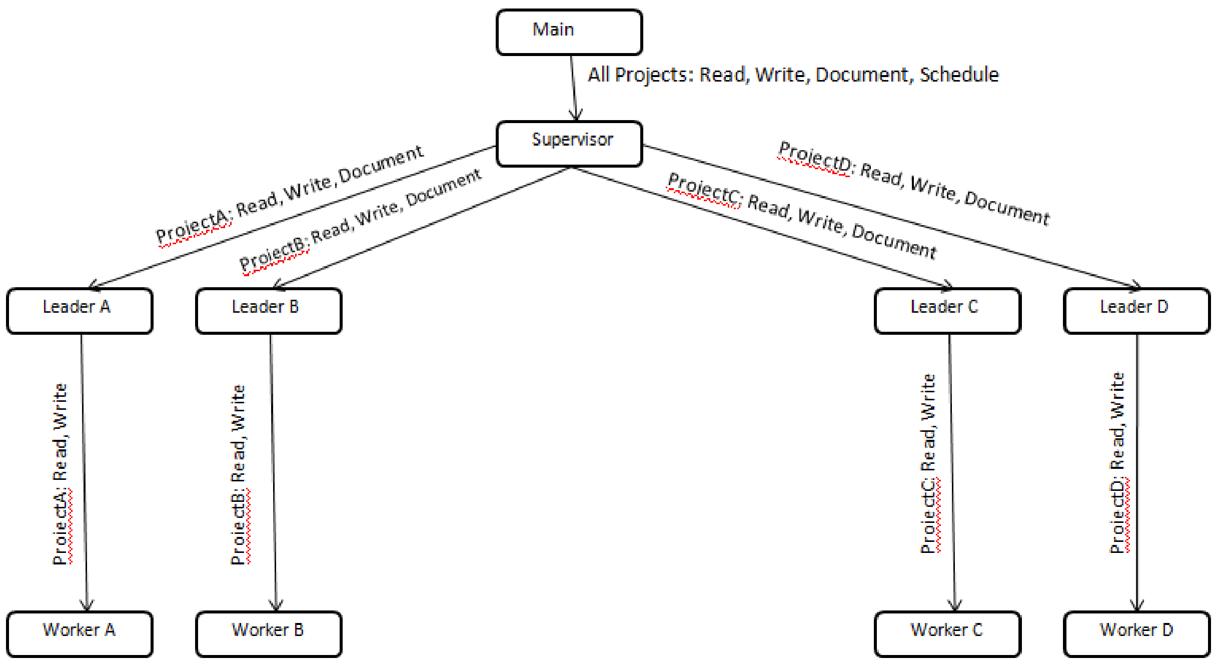
Chapter 6: Case Study

A private social network is incorporated into a company's project development process. The social network account will store all project related materials into its system. The roles within the network are given permissions that allow access and interact with those resources. The permissions in each role are determined by what permissions are given to them by a superior role. The organizational hierarchy of the project team has the supervisor setting the permissions to each of the section leaders, and the section leaders determine what permissions to each of their individual workers.

The permissions each role should have are based upon what tasks a person with that role must complete. A worker on a project must be able to at least read and write to their section's project resource, but never to any other. Section leaders not only have the same resource permissions of their workers, but are also the only ones who are able to create documentation for their project resource. Just like their workers, section leaders should never be able to interact with any other project but the one they are assigned to. Unlike all the other roles, the supervisor is the only one with all of the permissions to all of the projects stored inside of the account, meaning that he/she is able to execute section leader type commands on any project stored in the social network account. The supervisor is also the role that determines which project resource each section leader is allowed to work on and is the only role that can set the schedule to those projects. Since the supervisor role has so much influence upon the account, there should only be one user given the supervisor role.

Rules in the policy which applies to this system include:

1. There are four permissions for the four possible commands a user can execute upon the resources of the social network account. This includes Read, Write, Document, and Schedule.
2. A user of the supervisor role must have all of the root permissions to all of the project resources.
3. The supervisor determines which project resource each section leader and workers are able to have permissions to.
4. A user of a section leader role must have the permissions to Read, Write, and Document for their assigned project resource and should only be able to assign workers permissions to the same project.
5. A section leader should never be able to execute commands to any other resource but the one assigned to them by the supervisor role.
6. A user of the worker role must be able to Read and Write to a project resource. Which project resource is determined by which section leader is their superior.
7. Workers should never be able to execute commands to any other project that was not assigned to them. In essence, workers can only have permissions to the same project as their section leader.
8. If a role has a superior role assigning it permissions and resources, these two roles must not have the same set of permissions as this makes one of the roles redundant.
9. When assigning roles to a user, that user must never be assigned roles that will grant them access to more than one resource. The only exception to this rule is if the user is also given the supervisor role.
10. There should only be one user my log in with the supervisor role.



Main
All Projects: Read, Write, Document, Schedule
Supervisor
ProjectA: Read, Write, Document
ProjectB: Read, Write, Document
ProjectD: Read, Write, Document
ProjectC: Read, Write, Document
Leader A
Leader B
Leader C
Leader D
ProjectA: Read, Write
ProjectB: Read, Write
ProjectC: Read, Write
ProjectD: Read, Write
Worker A
Worker B
Worker C
Worker D

Figure 8: Model Representation of the Case Study.

6.1 Building the Model

Previously in the model checking chapter, it was mentioned that a common difficultly in constructing the model is the sheer size of the model itself. For a social network account, the factor that may cause a model to reach an encumbering size is the number of roles and permissions within the account and the assigning of those permissions to each role. Though this case study has only nine roles and four permissions to four different resources, it is possible for a social network to have any number of roles, permissions and resources. Creating a model by hand can become a lengthy process with human error contributing to the total time to finish the model. In order to shorten the amount of time needed to build the model in the case study and lower the opportunity for human error to occur, a Java code generator program was used.

As in the example model, the case study has a separate module to represent each role in the social network account. These role modules all follow the same outline. First, they are given permissions from a superior role. Second, they declare a user variable to receive these permissions and interact with the resources. If the role modules have subordinate roles, they create permission arrays for each role, in the VAR section, and set the values to the arrays in the ASSIGN section. The arrays are then sent to the subordinate role type variables. Finally after the ASSIGN section, the SPEC section contains any property specification that is needed in the role module.

The process just described is repeated for each role module that exists within the model and its uniformity can be used to easily build the model by the code generator program that was developed. The program begins by asking how many resources there will be and what permissions users will have to those resources. Next the program asks how many root roles there will be and what are they called. In this case study there is only one root, the supervisor role, but other models may require more. For each root role that exists, the program will ask which permissions to each resource the role will be assigned. An opportunity is then presented where the user is allowed to add any number of specifications to the SPEC section of the module. Afterwards, the program asks how many subordinate roles each role has and repeats the above process until the roles no longer have subordinate roles.

Once all of the model questions of the code generator have been answer, the program will then build the described model into a .smv file. This file will contain all of the modules needed to represent the social network account with all of the variables set to the needed values. Using the program as part of the development process allows for the ability to generate a model to represent any size social network account while minimizing the need to tediously manually write all of the modules and the variables contained within them.

6.2 Overview of the SMV Program

```
MODULE main

VAR

projectA :  Resource();

projectB :  Resource();

projectC :  Resource();

projectD :  Resource();
```

Starting in the main module once again, the variables of the resources are declared with the name of the project they will represent, A to D, and with a type specifier of Resource. Resource is another module that represents the resources, and the commands that can be executed upon the resource, that exist within the model. ProjectA through projectD are thus instances of the Resource module.

```
    supervisorPermA : array 0..3 of boolean;

    supervisorPermB : array 0..3 of boolean;

    supervisorPermC : array 0..3 of boolean;

    supervisorPermD : array 0..3 of boolean;
```

In our case study, the main module begins by passing the four resources to the supervisor role and a set of permission arrays for each resource. Therefore four Boolean arrays are declared whose elements will be used to determine what permissions the supervisor role will have for each of the Resources. Thus supervisorPermA contains what permissions the supervisor role has for projectA, supervisorPermB for projectB, supervisorPermC for projectC, and supervisorPermD for projectD. The arrays' sizes are set in correlation to the possible commands the user can send to the resources with zero for Read, one for Write, two for Document, and three for Schedule. So if supervisorPermC[2] is equal to true, then the supervisor role is allowed to execute the Document command upon projectC.

```
ASSIGN

supervisorPermA[0] := TRUE;

supervisorPermA[1] := TRUE;

supervisorPermA[2] := TRUE;

supervisorPermA[3] := TRUE;

supervisorPermB[0] := TRUE;

supervisorPermB[1] := TRUE;

supervisorPermB[2] := TRUE;

supervisorPermB[3] := TRUE;

supervisorPermC[0] := TRUE;
```

```
supervisorPermC[1] := TRUE;

supervisorPermC[2] := TRUE;

supervisorPermC[3] := TRUE;

supervisorPermD[0] := TRUE;

supervisorPermD[1] := TRUE;

supervisorPermD[2] := TRUE;

supervisorPermD[3] := TRUE;
```

In our case study, the supervisor role is to have all of the permission to every project resource within the network account. Therefore, all the elements in the permission arrays are set to true.

```
role1 : Supervisor(projectA, projectB, projectC, projectD, supervisorPermA, supervisorPermB, supervisorPermC, supervisorPermD);
```

The resources and the now set permission arrays are then used as parameters by the variable role1, which is declared as type Supervisor.

6.3 Modeling the Social Network Account Resources

```
MODULE Resource()

VAR

state :  {Wait, Read, Write, Document, Schedule};
```

```
ASSIGN

init(state) := {Wait};
```

The resource module follows the same outline as the model in the example. The enumerated variable state is declared with the values of Wait, Read, Write, Document, and Schedule for the possible commands users may send to the resource. The variable state is then initialized to the value of Wait.

6.4 Modeling the Social Network Account Roles

```
MODULE Supervisor(projectA, projectB, projectC, projectD, supervisorPermA,
supervisorPermB,supervisorPermC, supervisorPermD)

VAR

user1 : process User(projectA, projectB, projectC, projectD, supervisorPermA, supervisorPermB,
supervisorPermC, supervisorPermD);
```

The only role module that is initialized in the main module is that of Supervisor, which takes as parameters the instances of the network resources paired with permission arrays that determine what commands the user with this role will be allowed to execute. All of these variables are then given to the user1 variable which models a user, with a supervisor role, interacting with the system resources. User1 is declared with its type specifier as the User module and uses the same parameters given to the role by main as arguments. In the case study, the supervisor dictates the permissions assigned to the four section leaders. Thus the supervisor module will need a set of Boolean permission arrays for each subordinate role to the supervisor.

```
sectApermA : array 0..3 of boolean;

sectApermB : array 0..3 of boolean;

sectApermC : array 0..3 of boolean;

sectApermD : array 0..3 of boolean;


sectBpermA : array 0..3 of boolean;

sectBpermB : array 0..3 of boolean;

sectBpermC : array 0..3 of boolean;

sectBpermD : array 0..3 of boolean;


sectCpermA : array 0..3 of boolean;

sectCpermB : array 0..3 of boolean;

sectCpermC : array 0..3 of boolean;

sectCpermD : array 0..3 of boolean;


sectDpermA : array 0..3 of boolean;

sectDpermB : array 0..3 of boolean;
```

```
sectDpermC : array 0..3 of boolean;

sectDpermD : array 0..3 of boolean;
```

In the Var section, four by four sets of permission arrays are declared. The sixteen arrays equate to the need of four separate roles needing permission arrays to the four individual resources. To help differentiate which arrays go to which roles, the variable names are based upon the section they will be sent to and the resource it is for. For example, sectDpermA refers to it belonging to the section leader D and that it is the permission array to project resource A. The elements to each array must then be set based on the permissions each section leader requires in the case study.

```
ASSIGN

sectApermA[0] := TRUE;

sectApermA[1] := TRUE;

sectApermA[2] := TRUE;

sectApermA[3] := FALSE;


sectApermB[0] := FALSE;
```

```
sectApermB[1] := FALSE;

sectApermB[2] := FALSE;

sectApermB[3] := FALSE;


sectApermC[0] := FALSE;

sectApermC[1] := FALSE;

sectApermC[2] := FALSE;

sectApermC[3] := FALSE;


sectApermD[0] := FALSE;

sectApermD[1] := FALSE;

sectApermD[2] := FALSE;

sectApermD[3] := FALSE;
```

The leader of section A must be able use the commands of Read, Write, and Document to only project resource A. Thus the first three elements of array sectApermA are set to true and all the other resource permission arrays are set to false. The Schedule command is only available to the role of Supervisor and thus element four of the sectApermA is false. The permission arrays for the other sections follow the same pattern. Each section will only have the values of the first three elements if their required project resource set to true and all others will be false. Thus section leader B will have the permissions for project B, leader C for project C, and leader D for project D.

```
sectBpermA[0] := FALSE;

sectBpermA[1] := FALSE;

sectBpermA[2] := FALSE;

sectBpermA[3] := FALSE;


sectBpermB[0] := TRUE;

sectBpermB[1] := TRUE;

sectBpermB[2] := TRUE;

sectBpermB[3] := FALSE;


sectBpermC[0] := FALSE;

sectBpermC[1] := FALSE;

sectBpermC[2] := FALSE;

sectBpermC[3] := FALSE;


sectBpermD[0] := FALSE;
```

sectBpermD[1] := FALSE;

sectBpermD[2] := FALSE;

sectBpermD[3] := FALSE;

The leader of section B must be able use the commands of Read, Write, and Document to only project resource B. Thus the first three elements of array sectBpermB are set to true and all the other resource permission arrays are set to false. The Schedule command is only available to the role of Supervisor and thus element four of the sectBpermB is false.

sectCpermA[0] := FALSE;

sectCpermA[1] := FALSE;

sectCpermA[2] := FALSE;

sectCpermA[3] := FALSE;

sectCpermB[0] := FALSE;

sectCpermB[1] := FALSE;

sectCpermB[2] := FALSE;

sectCpermB[3] := FALSE;

sectCpermC[0] := TRUE;

sectCpermC[1] := TRUE;

sectCpermC[2] := TRUE;

sectCpermC[3] := FALSE;

sectCpermD[0] := FALSE;

sectCpermD[1] := FALSE;

sectCpermD[2] := FALSE;

sectCpermD[3] := FALSE;

The leader of section C must be able use the commands of Read, Write, and Document to only project resource C. Thus the first three elements of array sectCpermC are set to true and all the other resource permission arrays are set to false. The Schedule command is only available to the role of Supervisor and thus element four of the sectCpermC is false.

sectDpermA[0] := FALSE;

sectDpermA[1] := FALSE;

sectDpermA[2] := FALSE;

sectDpermA[3] := FALSE;

```
sectDpermB[0] := FALSE;

sectDpermB[1] := FALSE;

sectDpermB[2] := FALSE;

sectDpermB[3] := FALSE;


sectDpermC[0] := FALSE;

sectDpermC[1] := FALSE;

sectDpermC[2] := FALSE;

sectDpermC[3] := FALSE;


sectDpermD[0] := TRUE;

sectDpermD[1] := TRUE;

sectDpermD[2] := TRUE;

sectDpermD[3] := FALSE;
```

The leader of section D must be able use the commands of Read, Write, and Document to only project resource D. Thus the first three elements of array sectDpermD are set to true and all the other resource permission arrays are set to false. The Schedule command is only available to the role of Supervisor and thus element four of the sectDpermD is false.

userLeaderA : sectionLeaderA(projectA, projectB, projectC, projectD, sectApermA, sectApermB, sectApermC, sectApermD);

userLeaderB : sectionLeaderB(projectA, projectB, projectC, projectD, sectBpermA, sectBpermB, sectBpermC, sectBpermD);

userLeaderC : sectionLeaderC(projectA, projectB, projectC, projectD, sectCpermA, sectCpermB, sectCpermC, sectCpermD);

userLeaderD : sectionLeaderD(projectA, projectB, projectC, projectD, sectDpermA, sectDpermB, sectDpermC, sectDpermD);

After all sixteen elements to the permission arrays have been assigned, they are passed with the resources as parameters to the variables userLeaderA, userLeaderB, userLeaderC, and userLeaderD. These variables are declared with a type specifier of a sectionLeader_ module. Nominally, having a single type of module used for all four variables would be preferable for this program. However, the permissions passed from a section leader to their subordinates will be unique to their section and thus requires a separate role module for each section.

MODULE sectionLeaderA(projectA, projectB, projectC, projectD, sectApermA, sectApermB,sectApermC, sectApermD)

VAR

userLeaderA : process User(projectA, projectB, projectC, projectD, sectApermA, sectApermB, sectApermC, sectApermD);

workerpermA : array 0..3 of boolean;

```
workerpermB : array 0..3 of boolean;

workerpermC : array 0..3 of boolean;

workerpermD : array 0..3 of boolean;
```

Module sectionLeaderA receives the project resources and permission array variables from the supervisor role and passes them on to the userLeaderA variable that represents a user with the leaderA role sending commands to the resources. In the case study, each section leader also has a subordinate worker whose permissions they must arrange for their role. The VAR section thus includes a set of permission arrays to be passed for the workerA role.

```
ASSIGN

workerpermA[0]    := TRUE;

workerpermA[1]    := TRUE;

workerpermA[2]    := FALSE;

workerpermA[3]    := FALSE;
```

The worker for section leader A is only to be allowed to Read and Write for projectA and therefore, only the lowest two elements of workerpermA are set to true. Remaining elements to this array and all other arrays must then contain false since this worker should not be allowed to send any of those commands to the other network projects.

```
workerForA : workerA(projectA, projectB, projectC, projectD, workerpermA, workerpermB,
workerpermC, workerpermD);
```

The new permission arrays and resources will then be used as parameters for the workerForA variable, which is an instance of the workerA module, in the VAR section.

The modules sectionLeaderB, sectionLeaderC, and sectionLeaderD all have a similar layout to sectionLeaderA above. The differences are the permission array settings for their workers. The workers for sectionLeaderB only have the first two elements to projectB set to true, those of sectionLeaderC have the two elements to projectC and sectionLeaderD's workers have the two elements for projectD.

```
MODULE workerA(projectA, projectB, projectC, projectD, workerpermA,
workerpermB,workerpermC, workerpermD)

VAR

userWorkerA : process User(projectA, projectB, projectC, projectD, workerpermA,
workerpermB, workerpermC, workerpermD);
```

Since none of the workers have a subordinate role to set permissions for, the module will just take the parameters sent to it by the sectionLeader modules and use them as parameters for the userWorker variable that is an instance of the User module.

6.5 Modeling User Interaction with the Project Resources

```
MODULE User(projectA, projectB, projectC, projectD, permA, permB, permC, permD)
```

In the case study, users will be interacting with four different project resources with the commands they are allowed to send based upon the values within their given permission array elements. Thus the User module will take as parameters the four account resources, and the permission arrays that were sent from a role module.

```
VAR

myCommandA : { Wait, Read, Write, Document, Schedule};

myCommandB : { Wait, Read, Write, Document, Schedule};

myCommandC : { Wait, Read, Write, Document, Schedule};

myCommandD : { Wait, Read, Write, Document, Schedule};



ASSIGN

init(myCommandA) := Wait;

init(myCommandB) := Wait;

init(myCommandC) := Wait;

init(myCommandD) := Wait;
```

Since there are four resources, there must be four myCommand variables declared in the VAR section and have their initial state set to Wait in the ASSIGN section of the module.

```
next(myCommandA) := case
```

```
                    (permA[0] = TRUE) & (permA[1] = FALSE) & (permA[2] = FALSE) &
(permA[3] = FALSE) : {Wait,Read};

                    (permA[0] = TRUE) & (permA[1] = TRUE) & (permA[2] = FALSE) &
(permA[3] = FALSE)  : {Wait,Read, Write};

                    (permA[0] = TRUE) & (permA[1] = TRUE) & (permA[2] = TRUE) &
(permA[3] = FALSE)  : {Wait,Read, Write,Document};

                    (permA[0] = TRUE) & (permA[1] = TRUE) & (permA[2] = TRUE) &
(permA[3] = TRUE) : {Wait,Read, Write,Document, Schedule};

                              TRUE   :  Wait;

                            esac;
```

The next function for the commands will once again be determined by a case expression. Unlike the previous example, the case study requires that whatever permission the user is given must also receive the lower permissions as well. That is accomplished in our model by arranging the cases so that each element has a case where it is true and only the lower elements are true too. Thus if the user has the Document permission, permA[3]=TRUE, then all the elements less than three must also be true in permA. The default statement is still the same and is used whenever the user does not have any permission array elements with a true value.

```
 next(ResourceA.state) :=
                    case
                    myCommandA != Wait : myCommandA;
                    TRUE   : Wait;
                    esac;
```

Once the myCommand variables are set, their values will be used to determine what the next state of the related resources will become.

Modeling the User Session Roles

In order to test the dynamic changes to a user’s role in the account, four users are modeled to each have a leader role, with one of the leaders also given the supervisor role, and its subordinate worker role. The user with the supervisor role is will also be granted all roles implemented by the social network account.

```
  simonRoles : array 0..8 of boolean;
  milesRoles : array 0..8 of boolean;
  sarahRoles : array 0..8 of boolean;
  buddyRoles : array 0..8 of boolean;
```

In VAR of the main module, the four named userRoles variables shown above are Boolean arrays of ten elements. If an element is true in the array, then the named user is allowed to login as that role. Elements zero through three correspond to the workerA, workerB, workerC, and workerD roles. Elements four through seven correspond to the leaderA, leaderB, leaderC, and leaderD roles. Finally, if element nine's value is set to true, it represents that the user is allowed to log in as the supervisor.

```
        simonRoles[0] := TRUE;

        simonRoles[1] := TRUE;

        simonRoles[2] := TRUE;

        simonRoles[3] := TRUE;

        simonRoles[4] := TRUE;

        simonRoles[5] := TRUE;

        simonRoles[6] := TRUE;

        simonRoles[7] := TRUE;

        simonRoles[8] := TRUE;



        milesRoles[0] := FALSE;

        milesRoles[1] := TRUE;

        milesRoles[2] := FALSE;

        milesRoles[3] := FALSE;

        milesRoles[4] := FALSE;

        milesRoles[5] := TRUE;
```

```
milesRoles[6] := FALSE;

milesRoles[7] := FALSE;

milesRoles[8] := FALSE;



sarahRoles[0] := FALSE;

sarahRoles[1] := FALSE;

sarahRoles[2] := TRUE;

sarahRoles[3] := FALSE;

sarahRoles[4] := FALSE;

sarahRoles[5] := FALSE;

sarahRoles[6] := TRUE;

sarahRoles[7] := FALSE;

sarahRoles[8] := FALSE;



buddyRoles[0] := FALSE;

buddyRoles[1] := FALSE;

buddyRoles[2] := FALSE;
```

```
buddyRoles[3] := TRUE;

buddyRoles[4] := FALSE;

buddyRoles[5] := FALSE;

buddyRoles[6] := FALSE;

buddyRoles[7] := TRUE;

buddyRoles[8] := FALSE;
```

In the case study, Simon is selected to be the supervisor of the account while Miles, Sarah, and Buddy are given the roles of LeaderB, LeaderC, and LeaderD. In the above ASSIGN section, milesRoles second and fifth elements are set to true in order to assign Miles the worker and LeaderB roles. Sarah's array has only the third and sixth elements set to true in order for her to login as workerC or as LeaderC and all but the third and seventh elements are false for Buddy's roles array so that he may login as workerD or as LeaderD. Simon roles array is true in all elements since he is the supervisor of the account, and thus is allowed to login with any role of his choosing.

In the case study, the leader and supervisor roles are to only be used by one user at a time in the social network. Thus a semaphore is needed for each role.

```
semLA : semaphore();

semLB : semaphore();

semLC : semaphore();
```

```
            semLD : semaphore();

            semS : semaphore();
```

The above five variables are semaphores for each of the leader roles and one for the supervisor role. These variables are declared with a type specifier of the semaphore module that takes no parameters.

```
MODULE semaphore()

VAR

   sema : boolean;

   userName : { None, Simon, Miles, Sarah, Buddy};

ASSIGN

   init(sema) := FALSE;

   init(userName) := None;
```

The semaphore module is nearly identical to the one in the example model. The only difference is that the enumeration of userName was changed to the users represented in this system.

```
   name1 : { None, Simon, Miles, Sarah, Buddy};

   name2 : { None, Simon, Miles, Sarah, Buddy};

   name3 : { None, Simon, Miles, Sarah, Buddy};

   name4 : { None, Simon, Miles, Sarah, Buddy};
```

As in the example model, the name of each user must be stored into a variable in the VAR section of main. The value of that variable will be used by the semaphores to track which user activated the semaphore.

```
name1 := Simon;

name2 := Miles;

name3 := Sarah;

name4 := Buddy;
```

The ASSIGN section will then set each variable to a value that is not used by any of the other name variables.

```
   userSimon : process userSessionss( semLA, semLB, semLC, semLD, semS, simonRoles,
name1);

   userMiles : process userSessionss( semLA, semLB, semLC, semLD, semS, milesRoles,
name2);

   userSarah : process userSessionss( semLA, semLB, semLC, semLD, semS, sarahRoles,
name3);

   userBuddy : process userSessionss( semLA, semLB, semLC, semLD, semS, buddyRoles,
name4);
```

The main module will finally take all the semaphores, the pairs of the named roles arrays, the name variables, and uses them as parameters for a named user variable with a type specifier of the userSessions module.

```
MODULE userSessions(semLA, semLB, semLC, semLD, semS, givenRoles, myName)

VAR

  activeRole : {loggedOut, workerA, workerB, workerC, workerD, leaderA, leaderB, leaderC,
leaderD, supervisor};

ASSIGN

  init(activeRole) := {loggedOut};
```

The case study's userRoles module is shown above with a near same layout of parameters as the example model. It receives as parameters the semaphore, the userRoles variable, and the name variables from main. The enumerated variable activeRole, whose possible values are the roles of the social network account, represents the user's current role during a session. ActiveRole may also have the value loggedOut to represent the user leaving a session. The initial value of activeRole is set to loggedOut until the user's roles array can be examined.

```
          next(activeRole) := case

                                (givenRoles[8] & !semS.sema) : {supervisor};

                                (givenRoles[7] & !semLD.sema) : {leaderD};

                                (givenRoles[6] & !semLC.sema) : {leaderC};

                                (givenRoles[5] & !semLB.sema) : {leaderB};

                                (givenRoles[4] & !semLA.sema) : {leaderA};

                                (givenRoles[3]) : {workerD};
```

```
                                        (givenRoles[2]) : {workerC};

                                        (givenRoles[1]) : {workerB};

                                        (givenRoles[0]) : {workerA};

                                        TRUE : loggedOut;

                                        esac;
```

Similar to the example model's next functions of the userRoles module, the case study's next value of activeRole is based upon the values of the user's givenRoles array elements and the semaphore of the exclusive roles.

```
 next(semS.sema) := case

                    activeRole = supervisor : TRUE;

                    activeRole != supervisor & semS.userName = myName : FALSE;

                    TRUE : FALSE;

                   esac;



  next(semS.userName) := case

                    activeRole = supervisor : myName;

                    TRUE : None;

                   esac;
```

Following the example model's procedure, if the user log in with an exclusive role the semaphore's sema variable is set to True and the userName variable is equal to the value stored in myName. When the user changes roles, sema is changed back into false and the value of userName reverts to None.

6.6 Case Study Access Control Specifications

The access control properties for the case study will follow the same temporal logic formula of the example model from before. The main differences will be the fact that the number of permissions, the type of permissions, and number of resources whose access control must be verified.

```
SPEC AG ! (permA[0] = FALSE & myCommandA = Read)

SPEC AG ! (permA[1] = FALSE & myCommandA = Write)

SPEC AG ! (permA[2] = FALSE & myCommandA = Document)

SPEC AG ! (permA[3] = FALSE & myCommandA = Schedule)




SPEC AG ! (permB[0] = FALSE & myCommandB = Read)

SPEC AG ! (permB[1] = FALSE & myCommandB = Write)

SPEC AG ! (permB[2] = FALSE & myCommandB = Document)

SPEC AG ! (permB[3] = FALSE & myCommandB = Schedule)
SPEC AG ! (permC[0] = FALSE & myCommandC = Read)

SPEC AG ! (permC[1] = FALSE & myCommandC = Write)

SPEC AG ! (permC[2] = FALSE & myCommandC = Document)

SPEC AG ! (permC[3] = FALSE & myCommandC = Schedule)
```

```
SPEC AG ! (permD[0] = FALSE & myCommandD = Read)

SPEC AG ! (permD[1] = FALSE & myCommandD = Write)

SPEC AG ! (permD[2] = FALSE & myCommandD = Document)

SPEC AG ! (permD[3] = FALSE & myCommandD = Schedule)
```

Unlike the example model, all four project resources take the same type of commands from the users and thus the specifications for each permission array will be identical. Taking the specifications to the permission arrays element zero as an example, the specifications work by having the model checker verify that in "all global states", if the user does not have the Read permission to the project resource, then the user's myCommand cannot be set to Read.

```
                         TRUE   :  {Wait, Read};
```

As a counterexample, the default case of next(myCommandD) is changed to allow the next value to be either Wait or Read. Originally, the only users' able to have their myCommandD equal to Read were those with a True value in their permission array. These users were the ones assigned the roles of supervisor, workerD, and leaderD and this alteration will now allow any user to use the command upon projectD.

```
-- specification AG !(permD[0] = FALSE & myCommandD = Read) IN
role1.userLeaderA.workerForA.userWorkerA is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  role1.userLeaderA.workerpermA[0] = TRUE

  role1.userLeaderA.workerpermA[1] = TRUE

  role1.userLeaderA.workerpermA[2] = FALSE

  role1.userLeaderA.workerpermA[3] = FALSE

  role1.userLeaderA.workerpermB[0] = FALSE

  role1.userLeaderA.workerpermB[1] = FALSE

  role1.userLeaderA.workerpermB[2] = FALSE
```

```
role1.userLeaderA.workerpermB[3] = FALSE

role1.userLeaderA.workerpermC[0] = FALSE

role1.userLeaderA.workerpermC[1] = FALSE

role1.userLeaderA.workerpermC[2] = FALSE

role1.userLeaderA.workerpermC[3] = FALSE

role1.userLeaderA.workerpermD[0] = FALSE

role1.userLeaderA.workerpermD[1] = FALSE

role1.userLeaderA.workerpermD[2] = FALSE

role1.userLeaderA.workerpermD[3] = FALSE

role1.userLeaderA.workerForA.userWorkerA.myCommandA = Wait

role1.userLeaderA.workerForA.userWorkerA.myCommandB = Wait

role1.userLeaderA.workerForA.userWorkerA.myCommandC = Wait

role1.userLeaderA.workerForA.userWorkerA.myCommandD = Wait

-> Input: 1.2 <-

_process_selector_ = role1.userLeaderA.workerForA.userWorkerA

running = FALSE

role1.userSupervisor.running = FALSE
```

```
-> State: 1.2 <-

  role1.userLeaderA.workerForA.userWorkerA.myCommandD = Read
```

As expected the Read specifications for all myCommands, except myCommandD, was followed within the model. The supervisor, workerD, and LeaderD roles were the only ones who followed the specification since they originally were given the permission and would not have used the default case of next(myCommandD). The workerA, workerB, workerC, leaderA, leaderB, and leaderC roles failed the specification since the users of those roles were able to still send the Read command while not having the permission. As shown above, the user with the workerA role failed the specification since they were not given the Read permission from their superior, as shown in role1.userLeaderA.workerpermD[0] = FALSE from State1.1, but was still able to have myCommandD set to Read, as shown in the role1.userLeaderA.workerForA.userWorkerA.myCommandD = Read from state 1.2. In order for the model to pass these access control specifications, the user's ability to execute commands must be based upon the permissions given to them by their superiors.

6.7 Case Study Permission Hierarchy Specifications

In order for the case study model to follow permission hierarchy, when a user is given a permission to a resource they must also be given all the lower permissions as well. The users have four permissions that will grant them access to the four project resources in the social network account which are as follows in ascending order: Read, Write, Document, and Schedule. As an example of the permission hierarchy, if a user is granted the Document permission they must also be assigned Read and Write.

```
SPEC AG (  (permA[1]) -> (permA[0]) )
SPEC AG (  (permA[2]) -> (permA[0] & permA[1]) )

SPEC AG (  (permA[3]) -> (permA[0] & permA[1] & permA[2]) )



SPEC AG (  (permB[1]) -> (permB[0]) )

SPEC AG (  (permB[2]) -> (permB[0] & permB[1]) )

SPEC AG (  (permB[3]) -> (permB[0] & permB[1] & permB[2]) )



SPEC AG (  (permC[1]) -> (permC[0]) )

SPEC AG (  (permC[2]) -> (permC[0] & permC[1]) )

SPEC AG (  (permC[3]) -> (permC[0] & permC[1] & permC[2]) )



SPEC AG (  (permD[1]) -> (permD[0]) )

SPEC AG (  (permD[2]) -> (permD[0] & permD[1]) )

SPEC AG (  (permD[3]) -> (permD[0] & permD[1] & permD[2]) )
```

The above specifications are included in the User module to verify that permission hierarchy is followed in the model. Each set of specifications focuses on one resource permission array. As in the example model, each specification has the model checker verify that an element of a permission array being true implies that all lower elements of that array are true as well.

As a counterexample, LeaderB will assign workerB only the Write permission to projectB, permB[1], but without the Read permission, permB[0].

```
-- specification AG (permB[1] -> permB[0]) IN role1.userLeaderB.workerForB.userWorkerB is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  role1.userLeaderB.workerpermB[0] = FALSE

  role1.userLeaderB.workerpermB[1] = TRUE

  role1.userLeaderB.workerpermB[2] = FALSE

  role1.userLeaderB.workerpermB[3] = FALSE
```

The model checker reports that only workerB did not pass the permission hierarchy of its permB arrays. The trace shows that leader has set only workerB’s Write permission but not its Read permission, and thus violates the specification.

6.8 Case Study Minimum Duties Specification

All the roles of the case study are to be assigned the smallest set of permissions needed by the users of those roles to accomplish their required tasks. The case study requires that the supervisor role must be assigned all resource permissions, section leaders are only assigned the Read, Write, and Document permissions to one of the project resources and the workers must only have the Read and Write to the same project resources as their section leaders.

SPEC AG((workerpermA[1] & !workerpermA[2]) | (workerpermB[1] &!workerpermB[2]) | (workerpermC[1] & !workerpermC[2]) | (workerpermD[1] & !workerpermD[2]))

The above specification is added to the workerA, workerB, workerC, and workerD modules. The CTL logic has the model checker verify that in "all global states" one of the following conditions must be true: The Write permission of projectA resource, permA[1], is true while the Document permission of projectA resource,permA[2], is false; The Write permission of projectB resource, permB[1], is true while the Document permission of projectB resource, permB[2], is false; The Write permission of projectC resource, permC[1], is true while the Document permission of projectC resource,permC[2], is false; The Write permission of projectD resource, permD[1], is true while the Document permission of projectD resource, permD[2], is false.

Though the case study requires that both Read and Write are always given to the worker roles, the specification only needs the one element to verify the permission assignment. Permission hierarchy is also enforced by the model and thus if the user is given the Write permission, element one of the permission array, they must also have received the Read permission as well, element zero of the permission array.

As a counterexample, the workerC will be assigned the Read, Write, and Document permissions of the projectC resource.

```
-- specification AG ((((workerpermA[1] & !workerpermA[2]) | (workerpermB[1] &
!workerpermB[2])) | (workerpermC[1] & !workerpermC[2])) | (workerpermD[1] &
!workerpermD[2])) IN role1.userLeaderC.workerForC is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  role1.userLeaderC.workerpermC[0] = TRUE

  role1.userLeaderC.workerpermC[1] = TRUE

  role1.userLeaderC.workerpermC[2] = TRUE

  role1.userLeaderC.workerpermC[3] = FALSE
```

The output from the model checker states that the workerC role did not pass the minimum duties requirement. The trace shows that the LeaderC role has not only assigned Read, workerpermC[0], and Write, workerpermC[1], of projectC resource to the subordinate workerC role, but as also unnecessarily given the Document permission, workerpermC[2], as well. For the model to adhere to the minimum duties specification, the workers must only be assigned the required permissions, Read and Write.

SPEC AG((sectApermA[2] & !sectApermA[3]) | (sectApermB[2] &!sectApermB[3]) | (sectApermC[2] & !sectApermC[3]) | (sectApermD[2] & !sectApermD[3]))

The section leaders' minimum duties specification is the same formula as the workers, except that the elements are incremented by one to represent the lowest permission needed, element two for Document. Above is the specification implemented in the section leaderA module. As a counterexample, the supervisor will only be assigned the permissions of Read to the projectA resource to section leader A.

```
-- specification AG ((((sectApermA[2] & !sectApermA[3]) | (sectApermB[2] &
!sectApermB[3])) | (sectApermC[2] & !sectApermC[3])) | (sectApermD[2] & !sectApermD[3]))
IN role1.userLeaderA is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  role1.sectApermA[0] = TRUE

  role1.sectApermA[1] = FALSE

  role1.sectApermA[2] = FALSE

  role1.sectApermA[3] = FALSE
```

The printout of the model checker reports that role module of leaderA has failed the specification. The reason show the reason being that the supervisor has only given leaderA the Read permission, role1.sectApermA[0], while the minimum requirement for leaderA include the elements one and two. The supervisor's permission assignment must be corrected in order for the model pass the minimum duties specification.

SPEC AG((supervisorPermA[3]) & (supervisorPermB[3]) & (supervisorPermC[3]) & (supervisorPermD[3]))

The supervisor role's minimum duties specification is the simplest of all the roles. It requires all of the permissions to every project resource. Remembering that permission hierarchy is implemented by the model, all that is require of the specification is the highest permission, Schedule, of each resource permission array. If the role has the Schedule permission, then they must have also been assigned all the lower permissions as well. Thus the above specification requires that in "all global states" the value of true must be in the permission arrays for Schedule, element three, for every permission array.

As a counterexample, the supervisor role is not assigned the Schedule permission for projectC in the main module.

```
-- specification AG (((supervisorPermA[3] &supervisorPermB[3]) &supervisorPermC[3]) &supervisorPermD[3]) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample
```

```
-> State: 1.1 <-

supervisorPermA[0] = TRUE

supervisorPermA[1] = TRUE

supervisorPermA[2] = TRUE

supervisorPermA[3] = TRUE

supervisorPermB[0] = TRUE

supervisorPermB[1] = TRUE

supervisorPermB[2] = TRUE

supervisorPermB[3] = TRUE

supervisorPermC[0] = TRUE

supervisorPermC[1] = TRUE

supervisorPermC[2] = TRUE

supervisorPermC[3] = FALSE

supervisorPermD[0] = TRUE

supervisorPermD[1] = TRUE

supervisorPermD[2] = TRUE

supervisorPermD[3] = TRUE
```

The model checker printout reports that the supervisor's minimum duties specification was not upheld by the model. The following trace shows that the cause of the failure was due to supervisor's Schedule permission for projectC, supervisorPermC[3], was not set to true in main. In order for the model checker to pass the model, the supervisor role module must be granted the highest level permission to all project resources.

6.9 Case Study SSOD Specifications

SSOD requires that certain permissions must be exclusively assigned in order to prevent a user from having a pair of conflicting permissions. In this case study, the section leaders and their workers must only have permissions for one project resource in order to avoid a conflict. The only exception is that of the supervisor role who is allowed to have all of the permissions to every resource.

SPEC AG(((workerpermA[1]) -> AG (!workerpermB[1] & !workerpermC[1] & !workerpermD[1])) & ((workerpermB[1]) -> AG (!workerpermA[1] & !workerpermC[1] & !workerpermD[1])) & ((workerpermC[1]) -> AG (!workerpermA[1] & !workerpermB[1] & !workerpermD[1])) & ((workerpermD[1]) -> AG (!workerpermA[1] & !workerpermB[1] & !workerpermC[1])))

The above specification is the SSOD specification included in all of the worker role modules. The CTL statement has the model checker verify that in "all global states" of the worker role modules, one of the following is true. The first statement implies that should the element of workerpermA[1] being true, that in "all global states" it should not be possible for workerpermB[1] or workerpermC[1] or workerpermD[1] to be true. The reason why only the Write permission is used in the logic is because the minimum duties property requires that the highest permission workers should receive is Write. The second, third and fourth statements have the same formula but with a different resource permission array as the subject. This effectively ensures that should a worker role receive the Write permission of a project resource, they must only have permissions to that one project resource. To demonstrate, leaderB will assign the workerB role the Read and Write permissions to both projectB and projectC.

```
-- specification AG ((((workerpermA[1] -> AG ((!workerpermB[1] & !workerpermC[1]) &
!workerpermD[1])) & (workerpermB[1] -> AG ((!workerpermA[1] & !workerpermC[1]) &
!workerpermD[1]))) & (workerpermC[1] -> AG ((!workerpermA[1] & !workerpermB[1]) &
!workerpermD[1]))) & (workerpermD[1] -> AG ((!workerpermA[1] & !workerpermB[1]) &
!workerpermC[1]))) IN role1.userLeaderB.workerForB is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  role1.userLeaderB.workerpermB[0] = TRUE

  role1.userLeaderB.workerpermB[1] = TRUE

  role1.userLeaderB.workerpermC[0] = TRUE

  role1.userLeaderB.workerpermC[1] = TRUE
```

The model checker prints out that workerB has failed the SSOD specification and the trace provided shows that worker has the Read and Write permissions to both projectB and projectC resources. In order for the model to pass the specification, the worker must only be assigned the minimum duties permissions to one resource.

```
SPEC AG(( (sectApermA[2]) -> AG (!sectApermB[2] & !sectApermC[2] & !sectApermD[2]))
& ((sectApermB[2]) -> AG (!sectApermA[2] & !sectApermC[2] & !sectApermD[2])) &
((sectApermC[2]) -> AG (!sectApermA[2] & !sectApermB[2] & !sectApermD[2])) &
((sectApermD[2]) -> AG (!sectApermA[2] & !sectApermB[2] & !sectApermC[2]) ))
```

The section leaders' SSOD specification follows the same layout as their worker roles. Above is the specification included in the section leaderA module. All the other section leader modules' CTL statement will be identical except for the name of the permission array. However, the leaders' minimum duties requirements has element two of the permission arrays, Document, analyzed in the CTL statement. As a counterexample, leaderA will be assigned the Read, Write, and Document permissions to both projectA and projectD resources.

```
-- specification AG ((((sectApermA[2] -> AG ((!sectApermB[2] & !sectApermC[2]) &
!sectApermD[2])) & (sectApermB[2] -> AG ((!sectApermA[2] & !sectApermC[2]) &
!sectApermD[2]))) & (sectApermC[2] -> AG ((!sectApermA[2] & !sectApermB[2]) &
!sectApermD[2]))) & (sectApermD[2] -> AG ((!sectApermA[2] & !sectApermB[2]) &
!sectApermC[2]))) IN role1.userLeaderA is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-
```

```
role1.sectApermA[0] = TRUE

role1.sectApermA[1] = TRUE

role1.sectApermA[2] = TRUE

role1.sectApermD[0] = TRUE

role1.sectApermD[1] = TRUE

role1.sectApermD[2] = TRUE
```

The model checker reports that leaderA has failed the SSOD specification implemented within its module. The trace displays that leaderA has been assigned the minimum duties permissions to not only projectA but also projectD as well. In order for the model to pass the specification, each leader role must only have permissions to one resource.

In the model checking chapter, it was possible for a faulty model to pass its specifications when analyzed through the model checker. This is due to specifications that were not designed to properly discover the flaws of the model and have the model checker report it. This scenario is possible in the currently discussed SSOD specifications, which requires that users of certain roles are granted exclusive access to a single resource and that no others have the same access, except the user with the supervisor role. So far, the leader roles and worker roles have a SSOD specification in which the model will only pass if the roles are only assigned permissions to one resource. These specifications have worked to discover any arrangements that allow users of these roles access to multiple resources. The one flaw of the specification is that it only operates within the scope of the modules and does not know if different roles are given access to the same single resource. For example, if roles leaderB and leaderD were both given the section leader minimum duties permissions to only projectD resource the model would still pass. Even though the SSOD has obviously failed, since two different user roles have access to the same resource, the model checker would not discover the flaw with the current specification. This is because the scope of the specifications is only within the role modules they were implemented in and do not analyze the permissions to the other roles.

```
SPEC AG( ((sectApermA[2]) -> AG (!sectBpermA[2] & !sectCpermA[2] & !sectDpermA[2]))
&   ((sectBpermA[2]) -> AG (!sectApermA[2] & !sectCpermA[2] & !sectDpermA[2]))
&((sectCpermA[2]) -> AG (!sectApermA[2] & !sectBpermA[2] & !sectDpermA[2]))   &
((sectDpermA[2]) -> AG (!sectApermA[2] & !sectBpermA[2] & !sectCpermA[2]))  )
```

In order for the model checker to discover this breach in SSOD, the above of specification is implemented in the supervisor role module. Similar to the SSOD inside of each leader role module, this specification analyze the all of the section leaders' Document permission to the projectA resource. The CTL statement has the model checker verify that in "all global states" of the model only one of the following may be true:

- leaderA's being assigned the Document permission to projectA, sectApermA[2], implies that in "all global states" leaderB, leaderC, and leaderD have not been assigned the Document permission to projectA, sectBpermA[2], sectCpermA[2], sectDpermA[2].
- leaderB's being assigned the Document permission to projectA, sectBpermA[2], implies that in "all global states" leaderA, leaderC, and leaderD have not been assigned the Document permission to projectA, sectApermA[2], sectCpermA[2], sectDpermA[2].

- leaderC's being assigned the Document permission to projectA, sectCpermA[2], implies that in "all global states" leaderA, leaderB, and leaderD have not been assigned the Document permission to projectA, sectApermA[2], sectBpermA[2], sectDpermA[2].
- leaderD's being assigned the Document permission to projectA, sectDpermA[2], implies that in "all global states" leaderA, leaderB, and leaderC have not been assigned the Document permission to projectA, sectApermA[2], sectBpermA[2], sectDpermA[2].

The specification will thus ensure that each of the section leaders may have their minimum duties assigned to only one project resource with assurances that no other leader has been given access to the same project.

```
SPEC AG( ((sectApermB[2]) -> AG (!sectBpermB[2] & !sectCpermB[2] & !sectDpermB[2]))
&   ((sectBpermB[2]) -> AG (!sectApermB[2] & !sectCpermB[2] & !sectDpermB[2]))   &
((sectCpermB[2]) -> AG (!sectApermB[2] & !sectBpermB[2] & !sectDpermB[2]))   &
((sectDpermB[2]) -> AG (!sectApermB[2] & !sectBpermB[2] & !sectCpermB[2]))  )

SPEC AG( ((sectApermC[2]) -> AG (!sectBpermC[2] & !sectCpermC[2] & !sectDpermC[2]))
&   ((sectBpermC[2]) -> AG (!sectApermC[2] & !sectCpermC[2] & !sectDpermC[2]))   &
((sectCpermC[2]) -> AG (!sectApermC[2] & !sectBpermC[2] & !sectDpermC[2]))   &
((sectDpermC[2]) -> AG (!sectApermC[2] & !sectBpermC[2] & !sectCpermC[2]))  )

SPEC AG( ((sectApermD[2]) -> AG (!sectBpermD[2] & !sectCpermD[2] & !sectDpermD[2]))
&   ((sectBpermD[2]) -> AG (!sectApermD[2] & !sectCpermD[2] & !sectDpermD[2]))
&((sectCpermD[2]) -> AG (!sectApermD[2] & !sectBpermD[2] & !sectDpermD[2]))    &
((sectDpermD[2]) -> AG (!sectApermD[2] & !sectBpermD[2] & !sectCpermD[2]))  )
```

The other SSOD specifications following the first follow the same format, but focus on a different permission array. The first one above is ensuring that projectB is not accessed by multiple leaders, while the second and last are specifications to projectC and projectD.

As a counterexample, roles leaderC and leaderD will be granted the Read, Write, and Document permissions to only the resource projectD.

```
-- specification AG ((((sectApermD[2] -> AG ((!sectBpermD[2] & !sectCpermD[2]) &
!sectDpermD[2])) & (sectBpermD[2] -> AG ((!sectApermD[2] & !sectCpermD[2]) &
!sectDpermD[2]))) & (sectCpermD[2] -> AG ((!sectApermD[2] & !sectBpermD[2]) &
!sectDpermD[2]))) & (sectDpermD[2] -> AG ((!sectApermD[2] & !sectBpermD[2]) &
!sectCpermD[2]))) IN role1 is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  role1.sectCpermD[0] = TRUE
  role1.sectCpermD[1] = TRUE
  role1.sectCpermD[2] = TRUE
  role1.sectDpermD[0] = TRUE

  role1.sectDpermD[1] = TRUE

  role1.sectDpermD[2] = TRUE
```

The model checker reports that the specification involving the section leaders' access to projectD resource has failed. The trace shows the reason being that section leaderC and leaderD both have leader role access to projectD. In order for the model to pass the specifications, the supervisor must assign each leader roles access to only one leader role.

Though needed for the section leader roles, a similar set of specifications is not needed to verify that the worker roles have access to only one resource. This is due to the leader roles' role hierarchy specification, which ensures that whatever single resource the leaders have access to their subordinate roles may only have access to the same resource.

6.10 Case Study Role Hierarchy Specifications

In the case study, there are two sets of permission assignments in which role hierarchy must be established. When the supervisor assigns permissions to each of the section leader roles and when each section leader role assigns permissions to their subordinate worker roles. In order for the model to adhere to role hierarchy, the superior roles of the interaction must always have a higher level of permission than those of the lower roles.

```
SPEC AG( !(sectApermA[0]) -> AG !(workerpermA[0]) )

SPEC AG( (sectApermA[0] & !sectApermA[1]) -> AG !(workerpermA[0]) )
SPEC AG( (sectApermA[1] & !sectApermA[2]) -> AG !(workerpermA[1]) )

SPEC AG( (sectApermA[2] & !sectApermA[3]) -> AG !(workerpermA[2]) )

SPEC AG( (sectApermA[3]) -> AG !(workerpermA[3]))
```

In order for the model checker to discover a breach in the role hierarchy of the account roles between the section leader roles and their worker roles, the above of specifications are implemented in section leaderA role module. These specifications have the model checker verify that in "all global states" of the model, one of the following cases must be true:

- The leaderA role not being assigned the Read permission to projectA, !sectApermA[0], implies that in "all global states" its worker must not have the Read permission to projectA, !(workerpermA[0]).
- The leaderA role being assigned the Read permission to projectA, sectApermA[0], but not the Write permission, sectApermA[1], implies that in "all global states" its worker must not have the Read permission to projectA, !(workerpermA[0]).
- The leaderA role being assigned the Write permission to projectA, sectApermA[1], but not the Document permission, sectApermA[2], implies that in "all global states" its worker must not have the Write permission to projectA, !(workerpermA[1]).
- The leaderA role being assigned the Document permission to projectA, sectApermA[2], but not the Schedule permission, sectApermA[3], implies that in "all global states" its worker must not have the Document permission to projectA, !(workerpermA[2]).

- The leaderA role being assigned the Schedule permission to projectA, sectApermA[3], implies that in “all global states” its worker must not have the Schedule permission to projectA, !(workerpermA[3]).

The structure of the specification set assists in determining what the permission hierarchy level of the leader roles is. Which specification used is entirely based upon the permissions assigned to the section leaders from the supervisor. In this case study, the CTL statement where the section leaders receive the Document permission and not the Schedule permission is the specification that will be used. However, the rest are added anyway since the permissions of the roles may be changed during a project development and thus having all possible combinations of the role hierarchy included already will remove the needed to redevelop the role hierarchy specifications.

```
SPEC AG( !(sectApermB[0]) -> AG !(workerpermB[0]) )

SPEC AG( (sectApermB[0] & !sectApermB[1]) -> AG !(workerpermB[0]) )

SPEC AG( (sectApermB[1] & !sectApermB[2]) -> AG !(workerpermB[1]) )

SPEC AG( (sectApermB[2] & !sectApermB[3]) -> AG !(workerpermB[2]) )

SPEC AG( (sectApermB[3]) -> AG !(workerpermB[3]))


SPEC AG( !(sectApermC[0]) -> AG !(workerpermC[0]) )

SPEC AG( (sectApermC[0] & !sectApermC[1]) -> AG !(workerpermC[0]) )

SPEC AG( (sectApermC[1] & !sectApermC[2]) -> AG !(workerpermC[1]) )
```

SPEC AG((sectApermC[2] & !sectApermC[3]) -> AG !(workerpermC[2]))

SPEC AG((sectApermC[3]) -> AG !(workerpermC[3]))

SPEC AG(!(sectApermD[0]) -> AG !(workerpermD[0]))

SPEC AG((sectApermD[0] & !sectApermD[1]) -> AG !(workerpermD[0]))

SPEC AG((sectApermD[1] & !sectApermD[2]) -> AG !(workerpermD[1]))

SPEC AG((sectApermD[2] & !sectApermD[3]) -> AG !(workerpermD[2]))

SPEC AG((sectApermD[3]) -> AG !(workerpermD[3]))

The specification sets following the first follow the same CTL logic as before, except that each set analyzes a different permission array. The first is for the section leaderA's permissions to projectB, the second is to projectC, and the last is to projectD. The other section leader modules have a near identical set of specifications of their own. The only difference is the first half of the permission array name used to represent what section the array was sent to.

As a counterexample, the workerA role will be assigned the Read permission to the projectC resource by leaderA.

-- specification AG (!permC[0] -> AG !workerpermC[0]) IN role1.userLeaderA is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

```
-> State: 1.1 <-

  role1.sectApermA[0] = TRUE

  role1.sectApermA[1] = TRUE

  role1.sectApermA[2] = TRUE

  role1.sectApermA[3] = FALSE

  role1.sectApermB[0] = FALSE

  role1.sectApermB[1] = FALSE

  role1.sectApermB[2] = FALSE

  role1.sectApermB[3] = FALSE

  role1.sectApermC[0] = FALSE

  role1.sectApermC[1] = FALSE

  role1.sectApermC[2] = FALSE

  role1.sectApermC[3] = FALSE

  role1.sectApermD[0] = FALSE

  role1.sectApermD[1] = FALSE

  role1.sectApermD[2] = FALSE

  role1.sectApermD[3] = FALSE
```

```
role1.sectBpermA[0] = FALSE

role1.sectBpermA[1] = FALSE

role1.sectBpermA[2] = FALSE

role1.sectBpermA[3] = FALSE

role1.userLeaderA.workerpermA[0] = TRUE

role1.userLeaderA.workerpermA[1] = TRUE

role1.userLeaderA.workerpermA[2] = FALSE

role1.userLeaderA.workerpermA[3] = FALSE

role1.userLeaderA.workerpermB[0] = FALSE

role1.userLeaderA.workerpermB[1] = FALSE

role1.userLeaderA.workerpermB[2] = FALSE

role1.userLeaderA.workerpermB[3] = FALSE

role1.userLeaderA.workerpermC[0] = TRUE

role1.userLeaderA.workerpermC[1] = FALSE

role1.userLeaderA.workerpermC[2] = FALSE

role1.userLeaderA.workerpermC[3] = FALSE

role1.userLeaderA.workerpermD[0] = FALSE
```

```
 role1.userLeaderA.workerpermD[1] = FALSE

 role1.userLeaderA.workerpermD[2] = FALSE

 role1.userLeaderA.workerpermD[3] = FALSE
```

The model checker printout reports that the projectD role hierarchy specifications have failed in the leaderA module. More specifically, the specification that failed was the CTL statement requiring that a false value in the element zero of permC implying that workerpermC[0] must also be false. The trace shows that the State 1.1 is contrary to this as leaderA's permC[0] is false, shown by role1.sectApermC[0] = FALSE, while the workerA's permC[0] is true, shown by role1.userLeaderA.workerpermC[0] = TRUE. Either the leaderA's permission level to projectC resource must be increased or workerA's permission level must be decreased in order for the model to pass the specification.

```
SPEC AG( (!supervisorPermA[0]) -> AG !( (sectApermA[0]) | (sectBpermA[0]) |
(sectCpermA[0]) | (sectDpermA[0]) ) )

SPEC AG( (supervisorPermA[0] & !supervisorPermA[1]) -> AG !( (sectApermA[0]) |
(sectBpermA[0]) | (sectCpermA[0]) | (sectDpermA[0]) ) )

SPEC AG( (supervisorPermA[1] & !supervisorPermA[2]) -> AG !( (sectApermA[1]) |
(sectBpermA[1]) | (sectCpermA[1]) | (sectDpermA[1]) ) )

SPEC AG( (supervisorPermA[2] & !supervisorPermA[3]) -> AG !( (sectApermA[2]) |
(sectBpermA[2]) | (sectCpermA[2]) | (sectDpermA[2]) ) )

SPEC AG( (supervisorPermA[3]) -> AG !( (sectApermA[3]) | (sectBpermA[3]) |
(sectCpermA[3]) | (sectDpermA[3]) ) )
```

```
SPEC AG( (!supervisorPermB[0]) -> AG !( (sectApermB[0]) | (sectBpermB[0]) |
(sectCpermB[0]) | (sectDpermB[0]) ) )

SPEC AG( (supervisorPermB[0] & !supervisorPermB[1]) -> AG !( (sectApermB[0]) |
(sectBpermB[0]) | (sectCpermB[0]) | (sectDpermB[0]) ) )

SPEC AG( (supervisorPermB[1] & !supervisorPermB[2]) -> AG !( (sectApermB[1]) |
(sectBpermB[1]) | (sectCpermB[1]) | (sectDpermB[1]) ) )

SPEC AG( (supervisorPermB[2] & !supervisorPermB[3]) -> AG !( (sectApermB[2]) |
(sectBpermB[2]) | (sectCpermB[2]) | (sectDpermB[2]) ) )

SPEC AG( (supervisorPermB[3]) -> AG !( (sectApermB[3]) | (sectBpermB[3]) |
(sectCpermB[3]) | (sectDpermB[3]) ) )


SPEC AG( (!supervisorPermC[0]) -> AG !( (sectApermC[0]) | (sectBpermC[0]) |
(sectCpermC[0]) | (sectDpermC[0]) ) )

SPEC AG( (supervisorPermC[0] & !supervisorPermC[1]) -> AG !( (sectApermC[0]) |
(sectBpermC[0]) | (sectCpermC[0]) | (sectDpermC[0]) ) )

SPEC AG( (supervisorPermC[1] & !supervisorPermC[2]) -> AG !( (sectApermC[1]) |
(sectBpermC[1]) | (sectCpermC[1]) | (sectDpermC[1]) ) )

SPEC AG( (supervisorPermC[2] & !supervisorPermC[3]) -> AG !( (sectApermC[2]) |
(sectBpermC[2]) | (sectCpermC[2]) | (sectDpermC[2]) ) )
```

SPEC AG((supervisorPermC[3]) -> AG !((sectApermC[3]) | (sectBpermC[3]) | (sectCpermC[3]) | (sectDpermC[3])))

SPEC AG((!supervisorPermD[0]) -> AG !((sectApermD[0]) | (sectBpermD[0]) | (sectCpermD[0]) | (sectDpermD[0])))

SPEC AG((supervisorPermD[0] & !supervisorPermD[1]) -> AG !((sectApermD[0]) | (sectBpermD[0]) | (sectCpermD[0]) | (sectDpermD[0])))

SPEC AG((supervisorPermD[1] & !supervisorPermD[2]) -> AG !((sectApermD[1]) | (sectBpermD[1]) | (sectCpermD[1]) | (sectDpermD[1])))

SPEC AG((supervisorPermD[2] & !supervisorPermD[3]) -> AG !((sectApermD[2]) | (sectBpermD[2]) | (sectCpermD[2]) | (sectDpermD[2])))

SPEC AG((supervisorPermD[3]) -> AG !((sectApermD[3]) | (sectBpermD[3]) | (sectCpermD[3]) | (sectDpermD[3])))

The supervisor role's role hierarchy specification follows the same layout as that of the section leaders except that there are four sets of the five role hierarchy specification. Each is needed to verify that the supervisor role has a higher permission level than all of the section leader roles for each project resource. Examining the last specification, the CTL statement requires that the model checker verify that a true value in the supervisor's permD[3] implies that in "all global states" it should not be possible for section leaderA's permD[3], section leaderB's permD[3], section leaderC's permD[3], or section leaderD's permD[3] to be true. The section leaders' role hierarchy set only had a single worker permD inside of its formula since leader role had one subordinate role that it assigns permissions to. The supervisor role however has four subordinate roles, all of the leader roles, and thus its role hierarchy formula contains all four subordinate roles' permission arrays. This will instruct the model checker to compare the permission levels of all the supervisor's subordinates and verify that all are lower.

As a counter example, the supervisor role will only receive the Read permission to projectA from the main module.

```
-- specification AG ((supervisorPermA[0] & !supervisorPermA[1]) -> AG !(((sectApermA[0] |
sectBpermA[0]) | sectCpermA[0]) | sectDpermA[0])) IN role1 is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

supervisorPermA[0] = TRUE

supervisorPermA[1] = FALSE

supervisorPermA[2] = FALSE

supervisorPermA[3] = FALSE

  role1.sectApermA[0] = TRUE

  role1.sectApermA[1] = TRUE

  role1.sectApermA[2] = TRUE
  role1.sectApermA[3] = FALSE
```

The model checker prints that the supervisor's role hierarchy involving the permission array to projectA resource has failed. Specifically, the specification requiring that should the supervisor's highest permission equal Read, supervisorPermA[0] = TRUE, then the permission levels of the subordinate roles must be lower. This is contrary to displayed trace which shows that the section leaderA's highest permission level, sectApermA[2] = TRUE, is that of Document. In order for the model to pass the specification, either the supervisor's permission level to projectA resource must be set higher or section leaderA's permission level must be lowered.

6.11 Case Study DSOD Specifications

The DSOD property requires that users are not assigned roles in which they will have access to conflicting sets of permissions. In the case study, those roles are each section of the leader and worker roles to each other. The reason that each leader, and their subordinate workers, is separated from each other leaders and workers is because they must only to be given access to one resource. Thus when assigning users roles, they should only be granted roles that involve the same resource. For example, if a user is assigned the leaderA role then they must not be assigned any worker roles that are not for section A or any other leader roles. The only exception of this rule is that of supervisor, whose control of the account grants the user access to not only all permissions to all resources, but also all of the roles as well.

```
SPEC AG ( givenRoles[0] -> AG !(givenRoles[1] | givenRoles[2]| givenRoles[3] | givenRoles[5]
| givenRoles[6] | givenRoles[7]) | (givenRoles[8])  )
```

Similar to the example model's DSOD specification, each specification has the model verify that if the user was assigned that role, they should also not be assigned any role that conflicts with that role unless they are also assigned the supervisor role. The specification above requires that the model checker verify that in “all global” states, the user being assigned the workerA role, from the variable givenRoles[0], implies that in “all global” states one of following assignments must be true. First, the user is not assigned the roles of workerB, from variable givenRoles[1], workerC, from variable givenRoles[2], workerD, from variable givenRoles[3], leaderB, from variable givenRoles[5], leaderC, from variable givenRoles[6], or leaderD, from variable givenRoles[7]. Second, the user is assigned the supervisor role, from variable givenRoles[8]. The first case ensures that the user is not assigned any set of conflicting roles that will allow them access to different section resources and the second will tell the model checker to ignore the first case if the user has been assigned the supervisor role and thus may be granted any combination of roles in the account.

```
SPEC AG ( givenRoles[1] -> AG !(givenRoles[0] | givenRoles[2]| givenRoles[3] | givenRoles[4]
| givenRoles[6] | givenRoles[7]) | (givenRoles[8])  )

SPEC AG ( givenRoles[2] -> AG !(givenRoles[0] | givenRoles[1]| givenRoles[3] | givenRoles[4]
| givenRoles[5] | givenRoles[7]) | (givenRoles[8])  )

SPEC AG ( givenRoles[3] -> AG !(givenRoles[0] | givenRoles[1]| givenRoles[2] | givenRoles[4]
| givenRoles[6] | givenRoles[6]) | (givenRoles[8])  )

SPEC AG ( givenRoles[4] -> AG !(givenRoles[1] | givenRoles[2] | givenRoles[3] |
givenRoles[5] | givenRoles[6] | givenRoles[7]) | (givenRoles[8]) )
```

SPEC AG (givenRoles[5] -> AG !(givenRoles[0] | givenRoles[2] | givenRoles[7] | givenRoles[4] | givenRoles[6] | givenRoles[7]) | (givenRoles[8]))

SPEC AG (givenRoles[6] -> AG !(givenRoles[0] | givenRoles[1] | givenRoles[3] | givenRoles[4] | givenRoles[5] | givenRoles[7]) | (givenRoles[8]))

SPEC AG (givenRoles[7] -> AG !(givenRoles[0] | givenRoles[1] | givenRoles[2] | givenRoles[4] | givenRoles[5] | givenRoles[6]) | (givenRoles[8]))

Just like in the example model, there is a specification for each set of conflicting roles that exists in the model. The specifications after the first follow the same template for its CTL statement.

- If the user assigned the workerB role, givenRoles[1], they should not be assigned any other worker roles or any leader roles, except leaderB, unless they have also been assigned the supervisor roles.
- If the user assigned the workerC role, givenRoles[2], they should not be assigned any other worker roles or any leader roles, except leaderC, unless they have also been assigned the supervisor roles.
- If the user assigned the workerD role, givenRoles[3], they should not be assigned any other worker roles or any leader roles, except leaderD, unless they have also been assigned the supervisor roles.
- If the user assigned the leaderA role, givenRoles[4], they should not be assigned any other leader roles or any worker role, except workerA, unless they have also been assigned the supervisor roles.

- If the user assigned the leaderB role, givenRoles[5], they should not be assigned any other leader roles or any worker role, except workerB, unless they have also been assigned the supervisor roles.
- If the user assigned the leaderC role, givenRoles[6], they should not be assigned any other leader roles or any worker role, except workerC, unless they have also been assigned the supervisor roles.
- If the user assigned the leaderD role, givenRoles[7], they should not be assigned any other leader roles or any worker role, except workerD, unless they have also been assigned the supervisor roles.

As a counter example user Miles will be given the workerC role along with his already assigned roles of workerB and leaderB.

```
-- specification AG (givenRoles[1] -> (AG !((((((givenRoles[0] | givenRoles[2]) | givenRoles[3]) |
givenRoles[4]) | givenRoles[6]) | givenRoles[7]) | givenRoles[8])) IN userMiles is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-

  milesRoles[0] = FALSE

  milesRoles[1] = TRUE
  milesRoles[2] = TRUE

  milesRoles[3] = FALSE

  milesRoles[4] = FALSE

  milesRoles[5] = TRUE

  milesRoles[6] = FALSE

  milesRoles[7] = FALSE

  milesRoles[8] = FALSE

  milesRoles[9] = FALSE

-- specification AG (givenRoles[2] -> (AG !((((((givenRoles[0] | givenRoles[1]) | givenRoles[3]) |
givenRoles[4]) | givenRoles[5]) | givenRoles[7]) | givenRoles[8])) IN userMiles is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 2.1 <-

  milesRoles[0] = FALSE

  milesRoles[1] = TRUE

  milesRoles[2] = TRUE

  milesRoles[3] = FALSE
  milesRoles[4] = FALSE

  milesRoles[5] = TRUE

  milesRoles[6] = FALSE

  milesRoles[7] = FALSE

  milesRoles[8] = FALSE

  milesRoles[9] = FALSE

- specification AG (givenRoles[5] -> (AG !((((((givenRoles[0] | givenRoles[2]) | givenRoles[7]) |
givenRoles[4]) | givenRoles[6]) | givenRoles[7]) | givenRoles[8]))) IN userMiles is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 3.1 <-

  milesRoles[0] = FALSE

  milesRoles[1] = TRUE

  milesRoles[2] = TRUE

  milesRoles[3] = FALSE

  milesRoles[4] = FALSE

  milesRoles[5] = TRUE
```

```
milesRoles[6] = FALSE

milesRoles[7] = FALSE

milesRoles[8] = FALSE

milesRoles[9] = FALSE
```

The model checker reports in the printout that three of the DSOD specifications, workerB; workerC; and leaderB; failed within the model. As mentioned before, the specification failures helps with the determining why the user's assigned roles did not allow the model to pass. According to the traces, Miles was assigned the roles of workerB, as shown in variable milesRoles[1], workerC, as shown in variable milesRoles[2], and leader, as shown in variable milesRoles[5]. Since workerB and leaderB are not in conflict with each other, then the role workerC is the malefactor. In order for the model to pass the specification, any conflicting roles assigned to the user must be addressed.

```
SPEC AG ( ( (userSimon.activeRole = supervisor) -> AG !((userMiles.activeRole = supervisor) | (userSarah.activeRole = supervisor) | (userBuddy.activeRole = supervisor) )))
```

Along with the specifications asking the model checker to verify users' have not been assigned conflicting pairs of roles, another set of specifications must be included in the main module verifying that only one user is allowed to login as the supervisor. In the case study, the supervisor role must be exclusively assigned to only one user. The above specification has the model checker verify that if the user Simon is able to log in as the supervisor, then no other user can as well. The CTL statement translates that in "all global" states, user Simon's activeRole equal to supervisor implies that in “all global” states user Miles, Sarah, or Buddy can never log in as supervisor as well.

```
SPEC AG (  (  (userMiles.activeRole = supervisor) -> AG  !((userSimon.activeRole =
supervisor) | (userSarah.activeRole = supervisor) | (userBuddy.activeRole = supervisor) )))

SPEC AG (  (  (userSarah.activeRole = supervisor) -> AG  !((userSimon.activeRole =
supervisor) | (userMiles.activeRole = supervisor) | (userBuddy.activeRole = supervisor) )))

SPEC AG (  (  (userBuddy.activeRole = supervisor) -> AG  !((userSimon.activeRole =
supervisor) | (userMiles.activeRole = supervisor) | (userSarah.activeRole = supervisor) )))
```

Following user Simon’s supervisor DSOD specification is a similar specification for each user that exists in the model. Just like Simon’s, their specifications has the model checker verify that if a user is able to log in as the supervisor, no one else can.

As a counterexample, user Buddy will be assigned the supervisor role while user Simon is also still assigned as the supervisor.

```
-- specification AG (userSimon.activeRole = supervisor -> AG !((userMiles.activeRole =
supervisor | userSarah.activeRole = supervisor) | userBuddy.activeRole = supervisor))  is false

-- as demonstrated by the following execution sequence

Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 1.1 <-
```

```
  userSimon.activeRole = loggedOut
  userBuddy.activeRole = loggedOut
-> Input: 1.2 <-
  _process_selector_ = userSimon
  running = FALSE
  userBuddy.running = FALSE
  userSimon.running = TRUE
-> State: 1.2 <-
  userSimon.activeRole = supervisor
-> Input: 1.3 <-
  _process_selector_ = userBuddy
  userBuddy.running = TRUE
  userSimon.running = FALSE
-> State: 1.3 <-
  userBuddy.activeRole = supervisor
-- specification AG (userBuddy.activeRole = supervisor -> AG !((userSimon.activeRole =
supervisor | userMiles.activeRole = supervisor) | userSarah.activeRole = supervisor))  is false
-- as demonstrated by the following execution sequence
```

```
Trace Description: CTL Counterexample

Trace Type: Counterexample

-> State: 2.1 <-

  userSimon.activeRole = loggedOut

  userBuddy.activeRole = loggedOut

-> Input: 2.2 <-

  _process_selector_ = userBuddy

  running = FALSE

  userBuddy.running = TRUE

  userSimon.running = FALSE

-> State: 2.2 <-

  userBuddy.activeRole = supervisor

-> Input: 2.3 <-

  _process_selector_ = userSimon

  userBuddy.running = FALSE

  userSimon.running = TRUE

-> State: 2.3 <-
```

```
  userSimon.activeRole = supervisor
```

The model checker printout reports that users Simon and Buddy both failed the DSOD specification requiring that only one user be allowed to login as the supervisor. The first trace shows that from state 1.1 to state 1.3, user Simon's activeRole is equal to supervisor followed by user Buddy. The second trace reveals in state 2.1 to 2.3 that user Buddy's activeRole becomes supervisor followed by user Simon. In order for the model checker to pass the supervisor DSOD, only one user can ever be allowed to login as the supervisor.

Chapter 7: Conclusion

In this thesis, formal modeling and model checking are used to verify the security properties of social networks. These systems have become a major facilitator of communication for people around the world. A person, from any location, can interact with others in as personal a manner as they desire. Whether with an old acquaintance, or attempting to make a new one, social networks allow users to associate with people of any interest, taste, past or background.

To allow users to express themselves to the fullest, many social networks allow their clients to upload various contents onto their accounts. Any form of digital media, which is usually pictures, videos, and text documents, can be displayed by users to share with others inside of the network. Though all of the data is meant to be shared, not all users may wish for the general public to be able to access their account's stored contents. In order to support this desire, many social networks allow their users to not only dictate who can access their account resources, but also regulate the types of actions that the other users can perform upon those resources.

However, being that resource materials may consist of items of a sensitive nature, unauthorized access must be prevented at all costs. Thus social networks have implemented various security measures in order to ensure their users' resources are protected. The security measure covered in this paper is that of RBAC. This security policy requires that all users must receive a pre-defined role when operating inside of the system. This role is assigned a set of permissions that will influence what actions a user can perform upon system objects. Whenever a user attempts to interact with any object, their granted permissions are always analyzed to determine what commands they can execute. A social network's RBAC implementation allows for the creation of contact lists inside of the accounts that determines the assigning of roles to other users. Whenever anyone accesses any of the account materials, their name is compared to the entries of the contact list and returns a role prescribed for that user. The administrators of a social network account determine which permissions each of these roles should have. Depending upon which privileges were granted determines whether or not a user can write a comment, view the contents of the account, or even know that these objects even exist.

This thesis demonstrates how formal modeling and analysis of these security properties can assist in evaluating the effectiveness of the protection offered. Once a model of the social network and its RBAC protocols is created in the NuSMV input language, the NuSMV model checker can determine whether or not the RBAC properties hold within the model. This is accomplished by having each RBAC property translated into temporal logic specifications inside of the model. The model checker then explores every possible state of the social network model to determine if it is possible for the model to reach a state that violates any of the RBAC specifications. Should any breaches of the needed security protocol exists, the model will then output a trace showing which model components and specifications are the source of the conflict.

The case study provided involves a model of a private social network with multiple degrees of administrating both the project resources and the permissions to them. Which permissions are dictated to each role is based upon a superior role's prerogatives and the minimum duties that must be accomplished by each role. Though more than one project resource exists in the social network, each role is usually only allowed to be able to have access to one project resource and no other. The only exception to this case is the user with the supervisor role that created the account and thus should have all root permissions to every project that is uploaded.

The counter examples, in which each RBAC policy is circumvented, show that the model checker's CTL specifications can determine whether or not the desired access control protocols of the account were upheld. In each example, the model checker reports which specification failed and then prints a history trace showing how the model reached such an unacceptable state. The shown specification tells us which access control policy the user was able to ignore, and the trace tells the values and states of the modules that allowed the failure. When examined together, weaknesses that allowed the user unauthorized access can be quickly found. This can be seen in the first counter example, where users are allowed to execute commands while not having the permissions to do so. After the model checker examined the flawed model, the RBAC specifications reported that the model failed to uphold the necessary RBAC properties. The model checker then outputs traces to show how the user was able to bypass their given privileges. These pieces of evidence would lead one to the defective user module that allowed users to execute commands while not analyzing their permissions. During the development of a complex system, such as a social network, these counter examples can assist in creating test cases for the implementation. The inclusion of formal modeling and model checking can aid in fashioning an effective security policy that can be counted on to prevent unwarranted access.

REFERENCES

Hogben, G., (2007, October).Security Issues and Recommendations for Online Social Networks. *European Network and Security Agency*.

Bacon, J., Evans, D., Eyers, D., Migliavacc, M., Pietzuch, P., &Shand, B. (2010, July).Enforcing End-to-end Application Security in the Cloud. *www.cl.cam.ac.uk/~de239/mw10-cloud_security.pdf*

Migliavacca, M., Papagiannis, I., Eyers, D., Shand, B., Bacon, J., &Pietzuch, P. (2010, November) Distributed Middleware Enforcement of Event Flow Security Policy. *ACM/IFIP/USENIX 11th International Middleware Conference Paper*.

Carminati, B., & Ferrari, E. (2008, July). Privacy-Aware Collaborative Access Control in Web-Based Social Networks. *IFIP International Federation for Information Processing*.

Clarke, E., Grumberg, O., Long, D., (1992, January). Model Checking and abstraction. *In Proceeding of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*.

Clarke, E., Grumberg, O., Peled, D. (1999) Model Checking. *MIT Press*.

Huth, M., Ryan, M. (2004) *Logic in Computer Science: Modeling and Reasoning about Systems. Cambridge University Press.*

Baier, C.,Katoen, J. (2008) Principles of Model Checking. *MIT Press*.

Wasserman, S., &Faust, K.(1994)Social Network Analysis : Methods and Applications.*Cambridge University Press.*

Hogben, G.(1994) Security Issues of Social Networks.*ENISA Position Paper, W3C Workshop on the future of Social Networking.*

Sandu, R. (2001) Future Directions in Role-Based Access Control Models. *MMM-ACNS '01 Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security*, pp. 22 -26

Sandu, R., Coyne, E., Feinstein, H., &Youman, C.(1994) Role-Based Access Control Models. *Annual Computer Security Application Conference*.

Department of Defense.(1985, Dec) Department of Defense Trusted Computer System Evaluation Criteria DOD 5200-28-STD, The Orange Book. *Department of Defense*.

Osborn, S., Sandu, R., &Munawer, Q. (2000, May) Configuring Role-Based Access Control to Enforce Mandatory and Discretionary Access Control Policies. *ACM Transactions on Information and System Security, Vol 3, No. 2,* pp. 85-106.

Thion, R., &Coulonde, S. (2006) Modeling and Inferring on Role-Based Access Control Polices Using Data Dependencies. *Database and Expert Systems Applications*, pp. 914-923.

Castano, S., Fugini, M., Martella, G., &Samarati, P. (1995) *Database Security. Addison-Wesly*

Barker, S., &Stuckey, P. (2003) Flexible Access Control Policy Specification with Constraint Logic Programming. *ACM Transactions on Information System Security, Vol 6, Issue 4, 2003*

Bertino, E., Catania, B., Ferrari, E., & Perlasca, P. (2003) A Logical Framework for reasoning about access control models. *ACM Transactions on Information System Security, Vol 6, Issue 1, 2003,* pp. 71- 127

Cavada, R., Cimatti, A., Jochim, C., Keighren, G., Olivetti, E., Pistore, M., Roveri, M.,&Tchaltsev, A. (2010) NuSMV 2.5 User Manual. *http://www.cs.cmu.edu/˜modelcheck/smv/smvmanual.r2.2.ps.*

APPENDIX A: Example Model Coding

```
MODULE main
VAR
   movie : videoResource();
   movieReview : textFileResource();

   editorPermA : array 0..2 of boolean;
   editorPermB : array 0..1 of boolean;

   writerPermA : array 0..2 of boolean;
   writerPermB : array 0..1 of boolean;

  role1 : editorRole(movie, movieReview, editorPermA, editorPermB);
  role2 : writerRole(movie, movieReview, writerPermA, writerPermB);

   semEditor : semaphore();
   semWriter : semaphore();

   jamesRoles : array 0..2 of boolean;
   brianRoles : array 0..2 of boolean;
   jacobRoles : array 0..2 of boolean;
   willyRoles : array 0..2 of boolean;
   name1 : { None, James, Brian, Jacob, Willy};

   name2 : { None, James, Brian, Jacob, Willy};

   name3 : { None, James, Brian, Jacob, Willy};

   name4 : { None, James, Brian, Jacob, Willy};


   userJames : process userSessions( semEditor, semWriter, jamesRoles, name1);

   userBrian : process userSessions( semEditor, semWriter, brianRoles, name2);

   userJacob : process userSessions( semEditor, semWriter, jacobRoles, name3);

   userWilly : process userSessions( semEditor, semWriter, willyRoles, name4);


ASSIGN


  editorPermA[0] := TRUE;

  editorPermA[1] := TRUE;

  editorPermA[2] := TRUE;


  editorPermB[0] := FALSE;

  editorPermB[1] := FALSE;


  writerPermA[0] := FALSE;

  writerPermA[1] := FALSE;

  writerPermA[2] := FALSE;
```

```
writerPermB[0] := TRUE;

writerPermB[1] := TRUE;


jamesRoles[0] := TRUE;

jamesRoles[1] := TRUE;

jamesRoles[2] := FALSE;


brianRoles[0] := FALSE;

brianRoles[1] := FALSE;

brianRoles[2] := TRUE;


jacobRoles[0] := TRUE;

jacobRoles[1] := FALSE;

jacobRoles[2] := FALSE;


willyRoles[0] := TRUE;

willyRoles[1] := FALSE;

willyRoles[2] := FALSE;


name1 := James;

name2 := Brian;

name3 := Jacob;
```

```
  name4 := Willy;




-- DSOD Specification

SPEC AG (  (  (userJames.activeRole = editor) -> AG  !((userBrian.activeRole = editor) |
(userJacob.activeRole = editor) | (userWilly.activeRole = editor) )))

SPEC AG (  (  (userBrian.activeRole = editor) -> AG  !((userJames.activeRole = editor) |
(userJacob.activeRole = editor) | (userWilly.activeRole = editor) )))

SPEC AG (  (  (userJacob.activeRole = editor) -> AG  !((userJames.activeRole = editor) |
(userBrian.activeRole = editor) | (userWilly.activeRole = editor) )))

SPEC AG (  (  (userWilly.activeRole = editor) -> AG  !((userJames.activeRole = editor) |
(userBrian.activeRole = editor) | (userJacob.activeRole = editor) )))



MODULE writerRole(movie, movieReview, writerPermA, writerPermB)

VAR

   userWriter : process User(movie, movieReview, writerPermA, writerPermB);



-- Minimum Duties Specification

SPEC AG( (!writerPermA[0] & !writerPermA[1] & !writerPermA[2] ) & (writerPermB[1]) )



-- SSOD Specification

SPEC AG( (writerPermA[0] -> AG !writerPermB[0]) & (writerPermB[0] ->  AG
!writerPermB[0]) )
```

```
MODULE editorRole(movie, movieReview, editorPermA, editorPermB)

VAR


   userEditor : process User(movie, movieReview, editorPermA, editorPermB);




   internPermA : array 0..2 of boolean;

   internPermB : array 0..1 of boolean;




   role3 : internRole(movie, movieReview, internPermA, internPermB);

ASSIGN


  internPermA[0] := TRUE;

  internPermA[1] := FALSE;

  internPermA[2] := FALSE;


  internPermB[0] := FALSE;

  internPermB[1] := FALSE;


-- Minimum Duties Specification

SPEC AG( (editorPermA[2]) & (!editorPermB[0] & !editorPermB[1]) )
```

```
--Role Hierarchy Specification

SPEC AG( (!editorPermA[0]) -> AG !(internPermA[0]))

SPEC AG( (editorPermA[0] & !editorPermA[1]) -> AG !(internPermA[0]) )

SPEC AG( (editorPermA[1] & !editorPermA[2]) -> AG !(internPermA[1])  )

SPEC AG( (editorPermA[2]) -> AG !(internPermA[2]))



--Role Hierarchy Specification

SPEC AG( (!editorPermB[0]) -> AG !(internPermB[0]))

SPEC AG( (editorPermB[0] & !editorPermB[1]) -> AG !(internPermB[0]) )

SPEC AG( (editorPermB[1]) -> AG !(internPermB[1]))



-- SSOD Specification

SPEC AG( (editorPermA[0] -> AG !editorPermB[0]) & (editorPermB[0] ->  AG
!editorPermB[0]) )



MODULE internRole(movie, movieReview, internPermA, internPermB)

VAR

  userIntern : process User(movie, movieReview, internPermA, internPermB);



-- Minimum Duties Specification

SPEC AG( (internPermA[0] & !internPermA[1] & !internPermA[2]) & (!internPermB[0] &
!internPermB[1]))



-- SSOD Specification
```

```
SPEC AG( (internPermA[0] -> AG !internPermB[0]) | (internPermB[0] ->  AG
!internPermA[0]) )




MODULE videoResource()

VAR

   state : {Wait, Play, Copy, Delete};

ASSIGN

   init(state) := Wait;



MODULE textFileResource()

VAR

  state : {Wait, Read, Write};

ASSIGN

   init(state) := Wait;



MODULE User(movie, movieReview, permA, permB)

VAR



  myCommandA : {Wait, Play, Copy, Delete};

  myCommandB : {Wait, Read, Write};
```

```
ASSIGN
 init(myCommandA) := Wait;
 init(myCommandB) := Wait;

 next(myCommandA) :=        case
                     (permA[0] & !permA[1] & !permA[2]) : {Wait, Play};
                     (!permA[0] & permA[1] & !permA[2]) : {Wait, Copy};
                     (permA[0] & permA[1] & !permA[2]) : {Wait, Play, Copy};
                     (!permA[0] & !permA[1] & permA[2]) : {Wait, Delete};
                     (permA[0] & !permA[1] & permA[2]) : {Wait, Play, Delete};
                     (!permA[0] & permA[1] & permA[2]) : {Wait, Copy, Delete};
                     (permA[0] & permA[1] & permA[2]) : {Wait, Play, Copy, Delete};
                     TRUE   :  Wait;
                     esac;

 next(movie.state) :=  case
                    myCommandA != Wait : myCommandA;
                    TRUE   : Wait;
                    esac;

 next(myCommandB) := case
```

```
                              (permB[0] & !permB[1]) : {Wait, Read};
                              --(!permB[0] & permB[1]) : {Wait, Write};
                              (permB[0] & permB[1]) : {Wait, Read, Write};
                              TRUE   :  Wait;
                              esac;

  next(movieReview.state) := case
                             myCommandB != Wait : myCommandB;
                             TRUE   : Wait;
                             esac;
-- Access Control Specification
SPEC AG ! (permA[0] = FALSE & myCommandA = Play)
SPEC AG ! (permA[1] = FALSE & myCommandA = Copy)
SPEC AG ! (permA[2] = FALSE & myCommandA = Delete)

SPEC AG ! (permB[0] = FALSE & myCommandB = Read)
SPEC AG ! (permB[1] = FALSE & myCommandB = Write)

-- Permission Hierarchy Specification
SPEC AG (  (permA[1]) -> (permA[0]) )
SPEC AG (  (permA[2]) -> (permA[0] & permA[1]) )
SPEC AG (  (permB[1]) -> (permB[0]) )
```

```
MODULE userSessions(semEditor, semWriter, givenRoles, myName)

VAR

  activeRole : {loggedOut, intern, editor, writer};

ASSIGN

  init(activeRole) := {loggedOut};


  next(activeRole) := case

                        (givenRoles[2] & !semWriter.sema) : {writer};

                        (givenRoles[1] & !semEditor.sema) : {editor};

                        (givenRoles[0]) : {intern};

                        TRUE : loggedOut;

                        esac;




  next(semEditor.sema) := case

                  activeRole = editor : TRUE;

                  activeRole != editor & semEditor.userName = myName : FALSE;

                  TRUE : FALSE;

                esac;


  next(semEditor.userName) := case

                  activeRole = editor : myName;
```

```
                        TRUE : None;
                      esac;

  next(semWriter.sema) := case
                        activeRole = writer : TRUE;
                        activeRole != writer & semEditor.userName = myName : FALSE;
                        TRUE : FALSE;
                      esac;

  next(semWriter.userName) := case
                        activeRole = writer : myName;
                        TRUE : None;
                      esac;



-- DSOD Specification
SPEC AG ( givenRoles[1] -> AG !(givenRoles[2])  )
SPEC AG ( givenRoles[2] -> AG !(givenRoles[1])  )



MODULE semaphore()
VAR
  sema : boolean;
  userName : { None, James, Brian, Jacob, Willy};
ASSIGN
  init(sema) := FALSE;
  init(userName) := None;
```

APPENDIX B: Case Study SMV Coding

```
MODULE main

VAR

  projectA :  Resource();

  projectB :  Resource();

  projectC :  Resource();

  projectD :  Resource();


  supervisorPermA : array 0..3 of boolean;

  supervisorPermB : array 0..3 of boolean;

  supervisorPermC : array 0..3 of boolean;

  supervisorPermD : array 0..3 of boolean;


  role1 : Supervisor(projectA, projectB, projectC, projectD, supervisorPermA, supervisorPermB,
supervisorPermC, supervisorPermD);


   semLA : semaphore();

   semLB : semaphore();

   semLC : semaphore();

   semLD : semaphore();

   semS : semaphore();
   simonRoles : array 0..8 of boolean;

   milesRoles : array 0..8 of boolean;

   sarahRoles : array 0..8 of boolean;

   buddyRoles : array 0..8 of boolean;



   name1 : { None, Simon, Miles, Sarah, Buddy};

   name2 : { None, Simon, Miles, Sarah, Buddy};

   name3 : { None, Simon, Miles, Sarah, Buddy};

   name4 : { None, Simon, Miles, Sarah, Buddy};



   userSimon : process userSession( semLA, semLB, semLC, semLD, semS, simonRoles,
name1);

   userMiles : process userSession( semLA, semLB, semLC, semLD, semS, milesRoles, name2);

   userSarah : process userSession( semLA, semLB, semLC, semLD, semS, sarahRoles, name3);

   userBuddy : process userSession( semLA, semLB, semLC, semLD, semS, buddyRoles,
name4);



ASSIGN

  supervisorPermA[0] := TRUE;

  supervisorPermA[1] := TRUE;

  supervisorPermA[2] := TRUE;

  supervisorPermA[3] := TRUE;



  supervisorPermB[0] := TRUE;
```

```
supervisorPermB[1] := TRUE;

supervisorPermB[2] := TRUE;

supervisorPermB[3] := TRUE;


supervisorPermC[0] := TRUE;

supervisorPermC[1] := TRUE;

supervisorPermC[2] := TRUE;

supervisorPermC[3] := TRUE;


supervisorPermD[0] := TRUE;

supervisorPermD[1] := TRUE;

supervisorPermD[2] := TRUE;

supervisorPermD[3] := TRUE;


name1 := Simon;

name2 := Miles;

name3 := Sarah;

name4 := Buddy;


simonRoles[0] := TRUE;

simonRoles[1] := TRUE;

simonRoles[2] := TRUE;

simonRoles[3] := TRUE;
```

```
simonRoles[4] := TRUE;

simonRoles[5] := TRUE;

simonRoles[6] := TRUE;

simonRoles[7] := TRUE;

simonRoles[8] := TRUE;


milesRoles[0] := FALSE;

milesRoles[1] := TRUE;

milesRoles[2] := FALSE;

milesRoles[3] := FALSE;

milesRoles[4] := FALSE;

milesRoles[5] := TRUE;

milesRoles[6] := FALSE;

milesRoles[7] := FALSE;

milesRoles[8] := FALSE;


sarahRoles[0] := FALSE;

sarahRoles[1] := FALSE;

sarahRoles[2] := TRUE;

sarahRoles[3] := FALSE;

sarahRoles[4] := FALSE;

sarahRoles[5] := FALSE;

sarahRoles[6] := TRUE;
```

```
  sarahRoles[7] := FALSE;

  sarahRoles[8] := FALSE;



  buddyRoles[0] := FALSE;

  buddyRoles[1] := FALSE;

  buddyRoles[2] := FALSE;

  buddyRoles[3] := TRUE;

  buddyRoles[4] := FALSE;

  buddyRoles[5] := FALSE;

  buddyRoles[6] := FALSE;

  buddyRoles[7] := TRUE;

  buddyRoles[8] := FALSE;



-- DSOD specification

SPEC AG (  (  (userSimon.activeRole = supervisor) -> AG  !((userMiles.activeRole =
supervisor) | (userSarah.activeRole = supervisor) | (userBuddy.activeRole = supervisor) )))

SPEC AG (  (  (userMiles.activeRole = supervisor) -> AG  !((userSimon.activeRole =
supervisor) | (userSarah.activeRole = supervisor) | (userBuddy.activeRole = supervisor) )))

SPEC AG (  (  (userSarah.activeRole = supervisor) -> AG  !((userSimon.activeRole =
supervisor) | (userMiles.activeRole = supervisor) | (userBuddy.activeRole = supervisor) )))

SPEC AG (  (  (userBuddy.activeRole = supervisor) -> AG  !((userSimon.activeRole =
supervisor) | (userMiles.activeRole = supervisor) | (userSarah.activeRole = supervisor) )))




MODULE semaphore()
```

```
VAR

   -- If sema is False, a user is not using the exclusive role and may login with the role.  If sema is
True, a user is using the exclusive role and may not login with the role.

   sema : boolean;



   -- Variable used to store the name of the user that activated the semaphore.

   userName : { None, Simon, Miles, Sarah, Buddy};

ASSIGN

   init(sema) := FALSE;

   init(userName) := None;



MODULE Resource()

VAR

   state :  {Wait, Read, Write, Document, Schedule};



ASSIGN

  init(state) := {Wait};



MODULE Supervisor(projectA, projectB, projectC, projectD, supervisorPermA,
supervisorPermB,supervisorPermC, supervisorPermD)

VAR

  sectApermA : array 0..3 of boolean;

  sectApermB : array 0..3 of boolean;

  sectApermC : array 0..3 of boolean;
  sectApermD : array 0..3 of boolean;


  sectBpermA : array 0..3 of boolean;

  sectBpermB : array 0..3 of boolean;

  sectBpermC : array 0..3 of boolean;

  sectBpermD : array 0..3 of boolean;


  sectCpermA : array 0..3 of boolean;

  sectCpermB : array 0..3 of boolean;

  sectCpermC : array 0..3 of boolean;

  sectCpermD : array 0..3 of boolean;


  sectDpermA : array 0..3 of boolean;

  sectDpermB : array 0..3 of boolean;

  sectDpermC : array 0..3 of boolean;

  sectDpermD : array 0..3 of boolean;


  userSupervisor : process User(projectA, projectB, projectC, projectD, supervisorPermA,
supervisorPermB, supervisorPermC, supervisorPermD);


  userLeaderA : sectionLeaderA(projectA, projectB, projectC, projectD, sectApermA,
sectApermB, sectApermC, sectApermD);

  userLeaderB : sectionLeaderB(projectA, projectB, projectC, projectD, sectBpermA,
sectBpermB, sectBpermC, sectBpermD);
```

```
  userLeaderC : sectionLeaderC(projectA, projectB, projectC, projectD, sectCpermA,
sectCpermB, sectCpermC, sectCpermD);

  userLeaderD : sectionLeaderD(projectA, projectB, projectC, projectD, sectDpermA,
sectDpermB, sectDpermC, sectDpermD);



ASSIGN



  sectApermA[0] := TRUE;

  sectApermA[1] := TRUE;

  sectApermA[2] := TRUE;

  sectApermA[3] := FALSE;



  sectApermB[0] := FALSE;

  sectApermB[1] := FALSE;

  sectApermB[2] := FALSE;

  sectApermB[3] := FALSE;



  sectApermC[0] := FALSE;

  sectApermC[1] := FALSE;

  sectApermC[2] := FALSE;

  sectApermC[3] := FALSE;



  sectApermD[0] := FALSE;

  sectApermD[1] := FALSE;
  sectApermD[2] := FALSE;
  sectApermD[3] := FALSE;
--
  sectBpermA[0] := FALSE;
  sectBpermA[1] := FALSE;
  sectBpermA[2] := FALSE;
  sectBpermA[3] := FALSE;

  sectBpermB[0] := TRUE;
  sectBpermB[1] := TRUE;
  sectBpermB[2] := TRUE;
  sectBpermB[3] := FALSE;

  sectBpermC[0] := FALSE;
  sectBpermC[1] := FALSE;
  sectBpermC[2] := FALSE;
  sectBpermC[3] := FALSE;

  sectBpermD[0] := FALSE;
  sectBpermD[1] := FALSE;
  sectBpermD[2] := FALSE;
  sectBpermD[3] := FALSE;
--
```

```
  sectCpermA[0] := FALSE;

  sectCpermA[1] := FALSE;

  sectCpermA[2] := FALSE;

  sectCpermA[3] := FALSE;


  sectCpermB[0] := FALSE;

  sectCpermB[1] := FALSE;

  sectCpermB[2] := FALSE;

  sectCpermB[3] := FALSE;


  sectCpermC[0] := TRUE;

  sectCpermC[1] := TRUE;

  sectCpermC[2] := TRUE;

  sectCpermC[3] := FALSE;


  sectCpermD[0] := FALSE;

  sectCpermD[1] := FALSE;

  sectCpermD[2] := FALSE;

  sectCpermD[3] := FALSE;
--
  sectDpermA[0] := FALSE;

  sectDpermA[1] := FALSE;

  sectDpermA[2] := FALSE;
  sectDpermA[3] := FALSE;


  sectDpermB[0] := FALSE;

  sectDpermB[1] := FALSE;

  sectDpermB[2] := FALSE;

  sectDpermB[3] := FALSE;


  sectDpermC[0] := FALSE;

  sectDpermC[1] := FALSE;

  sectDpermC[2] := FALSE;

  sectDpermC[3] := FALSE;


  sectDpermD[0] := TRUE;

  sectDpermD[1] := TRUE;

  sectDpermD[2] := TRUE;

  sectDpermD[3] := FALSE;


-- Minimum Duties

SPEC AG( (supervisorPermA[3]) & (supervisorPermB[3]) & (supervisorPermC[3]) &
(supervisorPermD[3]))


--Role Hierarchy

SPEC AG( (!supervisorPermA[0]) -> AG !( (sectApermA[0]) | (sectBpermA[0]) |
(sectCpermA[0]) | (sectDpermA[0]) ) )
```

```
SPEC AG( (supervisorPermA[0] & !supervisorPermA[1]) -> AG !( (sectApermA[0]) |
(sectBpermA[0]) | (sectCpermA[0]) | (sectDpermA[0]) ) )

SPEC AG( (supervisorPermA[1] & !supervisorPermA[2]) -> AG !( (sectApermA[1]) |
(sectBpermA[1]) | (sectCpermA[1]) | (sectDpermA[1]) ) )

SPEC AG( (supervisorPermA[2] & !supervisorPermA[3]) -> AG !( (sectApermA[2]) |
(sectBpermA[2]) | (sectCpermA[2]) | (sectDpermA[2]) ) )

SPEC AG( (supervisorPermA[3]) -> AG !( (sectApermA[3]) | (sectBpermA[3]) |
(sectCpermA[3]) | (sectDpermA[3]) ) )



SPEC AG( (!supervisorPermB[0]) -> AG !( (sectApermB[0]) | (sectBpermB[0]) |
(sectCpermB[0]) | (sectDpermB[0]) ) )

SPEC AG( (supervisorPermB[0] & !supervisorPermB[1]) -> AG !( (sectApermB[0]) |
(sectBpermB[0]) | (sectCpermB[0]) | (sectDpermB[0]) ) )

SPEC AG( (supervisorPermB[1] & !supervisorPermB[2]) -> AG !( (sectApermB[1]) |
(sectBpermB[1]) | (sectCpermB[1]) | (sectDpermB[1]) ) )

SPEC AG( (supervisorPermB[2] & !supervisorPermB[3]) -> AG !( (sectApermB[2]) |
(sectBpermB[2]) | (sectCpermB[2]) | (sectDpermB[2]) ) )

SPEC AG( (supervisorPermB[3]) -> AG !( (sectApermB[3]) | (sectBpermB[3]) |
(sectCpermB[3]) | (sectDpermB[3]) ) )



SPEC AG( (!supervisorPermC[0]) -> AG !( (sectApermC[0]) | (sectBpermC[0]) |
(sectCpermC[0]) | (sectDpermC[0]) ) )

SPEC AG( (supervisorPermC[0] & !supervisorPermC[1]) -> AG !( (sectApermC[0]) |
(sectBpermC[0]) | (sectCpermC[0]) | (sectDpermC[0]) ) )

SPEC AG( (supervisorPermC[1] & !supervisorPermC[2]) -> AG !( (sectApermC[1]) |
(sectBpermC[1]) | (sectCpermC[1]) | (sectDpermC[1]) ) )

SPEC AG( (supervisorPermC[2] & !supervisorPermC[3]) -> AG !( (sectApermC[2]) |
(sectBpermC[2]) | (sectCpermC[2]) | (sectDpermC[2]) ) )

SPEC AG( (supervisorPermC[3]) -> AG !( (sectApermC[3]) | (sectBpermC[3]) |
(sectCpermC[3]) | (sectDpermC[3]) ) )
```

```
SPEC AG( (!supervisorPermD[0]) -> AG !( (sectApermD[0]) | (sectBpermD[0]) |
(sectCpermD[0]) | (sectDpermD[0]) ) )

SPEC AG( (supervisorPermD[0] & !supervisorPermD[1]) -> AG !( (sectApermD[0]) |
(sectBpermD[0]) | (sectCpermD[0]) | (sectDpermD[0]) ) )

SPEC AG( (supervisorPermD[1] & !supervisorPermD[2]) -> AG !( (sectApermD[1]) |
(sectBpermD[1]) | (sectCpermD[1]) | (sectDpermD[1]) ) )

SPEC AG( (supervisorPermD[2] & !supervisorPermD[3]) -> AG !( (sectApermD[2]) |
(sectBpermD[2]) | (sectCpermD[2]) | (sectDpermD[2]) ) )

SPEC AG( (supervisorPermD[3]) -> AG !( (sectApermD[3]) | (sectBpermD[3]) |
(sectCpermD[3]) | (sectDpermD[3]) ) )



-- Double SSOD 2

SPEC AG( ((sectApermA[2]) -> AG (!sectBpermA[2] & !sectCpermA[2] & !sectDpermA[2]))
&    ((sectBpermA[2]) -> AG (!sectApermA[2] & !sectCpermA[2] & !sectDpermA[2]))    &
((sectCpermA[2]) -> AG (!sectApermA[2] & !sectBpermA[2] & !sectDpermA[2]))    &
((sectDpermA[2]) -> AG (!sectApermA[2] & !sectBpermA[2] & !sectCpermA[2]))  )

SPEC AG( ((sectApermB[2]) -> AG (!sectBpermB[2] & !sectCpermB[2] & !sectDpermB[2]))
&    ((sectBpermB[2]) -> AG (!sectApermB[2] & !sectCpermB[2] & !sectDpermB[2]))    &
((sectCpermB[2]) -> AG (!sectApermB[2] & !sectBpermB[2] & !sectDpermB[2]))    &
((sectDpermB[2]) -> AG (!sectApermB[2] & !sectBpermB[2] & !sectCpermB[2]))  )

SPEC AG( ((sectApermC[2]) -> AG (!sectBpermC[2] & !sectCpermC[2] & !sectDpermC[2]))
&    ((sectBpermC[2]) -> AG (!sectApermC[2] & !sectCpermC[2] & !sectDpermC[2]))    &
((sectCpermC[2]) -> AG (!sectApermC[2] & !sectBpermC[2] & !sectDpermC[2]))    &
((sectDpermC[2]) -> AG (!sectApermC[2] & !sectBpermC[2] & !sectCpermC[2]))  )

SPEC AG( ((sectApermD[2]) -> AG (!sectBpermD[2] & !sectCpermD[2] & !sectDpermD[2]))
&    ((sectBpermD[2]) -> AG (!sectApermD[2] & !sectCpermD[2] & !sectDpermD[2]))    &
((sectCpermD[2]) -> AG (!sectApermD[2] & !sectBpermD[2] & !sectDpermD[2]))    &
((sectDpermD[2]) -> AG (!sectApermD[2] & !sectBpermD[2] & !sectCpermD[2]))  )



-- END OF SUPERVISOR
```

```
MODULE sectionLeaderA(projectA, projectB, projectC, projectD, sectApermA,
sectApermB,sectApermC, sectApermD)

VAR

  workerpermA : array 0..3 of boolean;

  workerpermB : array 0..3 of boolean;

  workerpermC : array 0..3 of boolean;

  workerpermD : array 0..3 of boolean;

  userLeaderA : process User(projectA, projectB, projectC, projectD, sectApermA, sectApermB,
sectApermC, sectApermD);

  workerForA : workerA(projectA, projectB, projectC, projectD, workerpermA, workerpermB,
workerpermC, workerpermD);



ASSIGN



   workerpermA[0]    := TRUE;

   workerpermA[1]    := TRUE;

   workerpermA[2]    := FALSE;

   workerpermA[3]    := FALSE;



   workerpermB[0]    := FALSE;

   workerpermB[1]    := FALSE;

   workerpermB[2]    := FALSE;
   workerpermB[3]    := FALSE;


   workerpermC[0]    := FALSE;

   workerpermC[1]    := FALSE;

   workerpermC[2]    := FALSE;

   workerpermC[3]    := FALSE;


   workerpermD[0]    := FALSE;

   workerpermD[1]    := FALSE;

   workerpermD[2]    := FALSE;

   workerpermD[3]    := FALSE;


-- Minimum Duties

SPEC AG( (sectApermA[2] & !sectApermA[3]) | (sectApermB[2] &!sectApermB[3]) |
(sectApermC[2] & !sectApermC[3]) | (sectApermD[2] & !sectApermD[3]))


-- Static Separation of Duties

SPEC AG(( (sectApermA[2]) -> AG (!sectApermB[2] & !sectApermC[2] & !sectApermD[2]))
& ((sectApermB[2]) -> AG (!sectApermA[2] & !sectApermC[2] & !sectApermD[2])) &
((sectApermC[2]) -> AG (!sectApermA[2] & !sectApermB[2] & !sectApermD[2])) &
((sectApermD[2]) -> AG (!sectApermA[2] & !sectApermB[2] & !sectApermC[2]) ))


--Role Hierarchy

SPEC AG( !(sectApermA[0]) -> AG !(workerpermA[0]) )

SPEC AG( (sectApermA[0] & !sectApermA[1]) -> AG !(workerpermA[0]) )
SPEC AG( (sectApermA[1] & !sectApermA[2]) -> AG !(workerpermA[1]) )

SPEC AG( (sectApermA[2] & !sectApermA[3]) -> AG !(workerpermA[2]) )

SPEC AG( (sectApermA[3]) -> AG !(workerpermA[3]))


SPEC AG( !(sectApermB[0]) -> AG !(workerpermB[0]) )

SPEC AG( (sectApermB[0] & !sectApermB[1]) -> AG !(workerpermB[0]) )

SPEC AG( (sectApermB[1] & !sectApermB[2]) -> AG !(workerpermB[1]) )

SPEC AG( (sectApermB[2] & !sectApermB[3]) -> AG !(workerpermB[2]) )

SPEC AG( (sectApermB[3]) -> AG !(workerpermB[3]))


SPEC AG( !(sectApermC[0]) -> AG !(workerpermC[0]) )

SPEC AG( (sectApermC[0] & !sectApermC[1]) -> AG !(workerpermC[0]) )

SPEC AG( (sectApermC[1] & !sectApermC[2]) -> AG !(workerpermC[1]) )

SPEC AG( (sectApermC[2] & !sectApermC[3]) -> AG !(workerpermC[2]) )

SPEC AG( (sectApermC[3]) -> AG !(workerpermC[3]))


SPEC AG( !(sectApermD[0]) -> AG !(workerpermD[0]) )

SPEC AG( (sectApermD[0] & !sectApermD[1]) -> AG !(workerpermD[0]) )

SPEC AG( (sectApermD[1] & !sectApermD[2]) -> AG !(workerpermD[1]) )

SPEC AG( (sectApermD[2] & !sectApermD[3]) -> AG !(workerpermD[2]) )

SPEC AG( (sectApermD[3]) -> AG !(workerpermD[3]))


----------------- End of sectionLeader A
```

```
MODULE sectionLeaderB(projectA, projectB, projectC, projectD, sectBpermA,
sectBpermB,sectBpermC, sectBpermD)

VAR

  workerpermA : array 0..3 of boolean;

  workerpermB : array 0..3 of boolean;

  workerpermC : array 0..3 of boolean;

  workerpermD : array 0..3 of boolean;

  userLeaderB : process User(projectA, projectB, projectC, projectD, sectBpermA, sectBpermB,
sectBpermC, sectBpermD);

  workerForB : workerB(projectA, projectB, projectC, projectD, workerpermA, workerpermB,
workerpermC, workerpermD);



ASSIGN

   workerpermA[0]    := FALSE;

   workerpermA[1]    := FALSE;

   workerpermA[2]    := FALSE;

   workerpermA[3]    := FALSE;



   workerpermB[0]    := TRUE;

   workerpermB[1]    := TRUE;

   workerpermB[2]    := FALSE;

   workerpermB[3]    := FALSE;
  workerpermC[0]    := FALSE;

  workerpermC[1]    := FALSE;

  workerpermC[2]    := FALSE;

  workerpermC[3]    := FALSE;


  workerpermD[0]    := FALSE;

  workerpermD[1]    := FALSE;

  workerpermD[2]    := FALSE;

  workerpermD[3]    := FALSE;


-- Minimum Duties

SPEC AG( (sectBpermA[2] & !sectBpermA[3]) | (sectBpermB[2] &!sectBpermB[3]) |
(sectBpermC[2] & !sectBpermC[3]) | (sectBpermD[2] & !sectBpermD[3]))


-- Static Separation of Duties

SPEC AG(( (sectBpermA[2]) -> AG (!sectBpermB[2] & !sectBpermC[2] & !sectBpermD[2])) &
((sectBpermB[2]) -> AG (!sectBpermA[2] & !sectBpermC[2] & !sectBpermD[2])) &
((sectBpermC[2]) -> AG (!sectBpermA[2] & !sectBpermB[2] & !sectBpermD[2])) &
((sectBpermD[2]) -> AG (!sectBpermA[2] & !sectBpermB[2] & !sectBpermC[2]) ))


--Role Hierarchy

SPEC AG( !(sectBpermA[0]) -> AG !(workerpermA[0]) )

SPEC AG( (sectBpermA[0] & !sectBpermA[1]) -> AG !(workerpermA[0]) )

SPEC AG( (sectBpermA[1] & !sectBpermA[2]) -> AG !(workerpermA[1]) )

SPEC AG( (sectBpermA[2] & !sectBpermA[3]) -> AG !(workerpermA[2]) )
SPEC AG( (sectBpermA[3]) -> AG !(workerpermA[3]))


SPEC AG( !(sectBpermB[0]) -> AG !(workerpermB[0]) )

SPEC AG( (sectBpermB[0] & !sectBpermB[1]) -> AG !(workerpermB[0]) )

SPEC AG( (sectBpermB[1] & !sectBpermB[2]) -> AG !(workerpermB[1]) )

SPEC AG( (sectBpermB[2] & !sectBpermB[3]) -> AG !(workerpermB[2]) )

SPEC AG( (sectBpermB[3]) -> AG !(workerpermB[3]))


SPEC AG( !(sectBpermC[0]) -> AG !(workerpermC[0]) )

SPEC AG( (sectBpermC[0] & !sectBpermC[1]) -> AG !(workerpermC[0]) )

SPEC AG( (sectBpermC[1] & !sectBpermC[2]) -> AG !(workerpermC[1]) )

SPEC AG( (sectBpermC[2] & !sectBpermC[3]) -> AG !(workerpermC[2]) )

SPEC AG( (sectBpermC[3]) -> AG !(workerpermC[3]))


SPEC AG( !(sectBpermD[0]) -> AG !(workerpermD[0]) )

SPEC AG( (sectBpermD[0] & !sectBpermD[1]) -> AG !(workerpermD[0]) )

SPEC AG( (sectBpermD[1] & !sectBpermD[2]) -> AG !(workerpermD[1]) )

SPEC AG( (sectBpermD[2] & !sectBpermD[3]) -> AG !(workerpermD[2]) )

SPEC AG( (sectBpermD[3]) -> AG !(workerpermD[3]))


----------------- End of sectionLeader B
```

```
MODULE sectionLeaderC(projectA, projectB, projectC, projectD, sectCpermA,
sectCpermB,sectCpermC, sectCpermD)

VAR

  workerpermA : array 0..3 of boolean;

  workerpermB : array 0..3 of boolean;

  workerpermC : array 0..3 of boolean;

  workerpermD : array 0..3 of boolean;

  userLeaderC : process User(projectA, projectB, projectC, projectD, sectCpermA, sectCpermB,
sectCpermC, sectCpermD);

  workerForC : workerC(projectA, projectB, projectC, projectD, workerpermA, workerpermB,
workerpermC, workerpermD);



ASSIGN

   workerpermA[0]    := FALSE;

   workerpermA[1]    := FALSE;

   workerpermA[2]    := FALSE;

   workerpermA[3]    := FALSE;



   workerpermB[0]    := FALSE;

   workerpermB[1]    := FALSE;

   workerpermB[2]    := FALSE;

   workerpermB[3]    := FALSE;



   workerpermC[0]    := TRUE;
  workerpermC[1]    := TRUE;

  workerpermC[2]    := FALSE;

  workerpermC[3]    := FALSE;



  workerpermD[0]    := FALSE;

  workerpermD[1]    := FALSE;

  workerpermD[2]    := FALSE;

  workerpermD[3]    := FALSE;



-- Minimum Duties

SPEC AG( (sectCpermA[2] & !sectCpermA[3]) | (sectCpermB[2] &!sectCpermB[3]) |
(sectCpermC[2] & !sectCpermC[3]) | (sectCpermD[2] & !sectCpermD[3]))



-- Static Separation of Duties

SPEC AG(( (sectCpermA[2]) -> AG (!sectCpermB[2] & !sectCpermC[2] & !sectCpermD[2])) &
((sectCpermB[2]) -> AG (!sectCpermA[2] & !sectCpermC[2] & !sectCpermD[2])) &
((sectCpermC[2]) -> AG (!sectCpermA[2] & !sectCpermB[2] & !sectCpermD[2])) &
((sectCpermD[2]) -> AG (!sectCpermA[2] & !sectCpermB[2] & !sectCpermC[2]) ))



--Role Hierarchy

SPEC AG( !(sectCpermA[0]) -> AG !(workerpermA[0]) )

SPEC AG( (sectCpermA[0] & !sectCpermA[1]) -> AG !(workerpermA[0]) )

SPEC AG( (sectCpermA[1] & !sectCpermA[2]) -> AG !(workerpermA[1]) )

SPEC AG( (sectCpermA[2] & !sectCpermA[3]) -> AG !(workerpermA[2]) )

SPEC AG( (sectCpermA[3]) -> AG !(workerpermA[3]))
SPEC AG( !(sectCpermB[0]) -> AG !(workerpermB[0]) )

SPEC AG( (sectCpermB[0] & !sectCpermB[1]) -> AG !(workerpermB[0]) )

SPEC AG( (sectCpermB[1] & !sectCpermB[2]) -> AG !(workerpermB[1]) )

SPEC AG( (sectCpermB[2] & !sectCpermB[3]) -> AG !(workerpermB[2]) )

SPEC AG( (sectCpermB[3]) -> AG !(workerpermB[3]))


SPEC AG( !(sectCpermC[0]) -> AG !(workerpermC[0]) )

SPEC AG( (sectCpermC[0] & !sectCpermC[1]) -> AG !(workerpermC[0]) )

SPEC AG( (sectCpermC[1] & !sectCpermC[2]) -> AG !(workerpermC[1]) )

SPEC AG( (sectCpermC[2] & !sectCpermC[3]) -> AG !(workerpermC[2]) )

SPEC AG( (sectCpermC[3]) -> AG !(workerpermC[3]))


SPEC AG( !(sectCpermD[0]) -> AG !(workerpermD[0]) )

SPEC AG( (sectCpermD[0] & !sectCpermD[1]) -> AG !(workerpermD[0]) )

SPEC AG( (sectCpermD[1] & !sectCpermD[2]) -> AG !(workerpermD[1]) )

SPEC AG( (sectCpermD[2] & !sectCpermD[3]) -> AG !(workerpermD[2]) )

SPEC AG( (sectCpermD[3]) -> AG !(workerpermD[3]))


----------------- End of sectionLeader C


MODULE sectionLeaderD(projectA, projectB, projectC, projectD, sectDpermA,
sectDpermB,sectDpermC, sectDpermD)
```

```
VAR

  workerpermA : array 0..3 of boolean;

  workerpermB : array 0..3 of boolean;

  workerpermC : array 0..3 of boolean;

  workerpermD : array 0..3 of boolean;

  userLeaderD : process User(projectA, projectB, projectC, projectD, sectDpermA, sectDpermB,
sectDpermC, sectDpermD);

  workerForD : workerD(projectA, projectB, projectC, projectD, workerpermA, workerpermB,
workerpermC, workerpermD);



ASSIGN

   workerpermA[0]    := FALSE;

   workerpermA[1]    := FALSE;

   workerpermA[2]    := FALSE;

   workerpermA[3]    := FALSE;



   workerpermB[0]    := FALSE;

   workerpermB[1]    := FALSE;

   workerpermB[2]    := FALSE;

   workerpermB[3]    := FALSE;



   workerpermC[0]    := FALSE;

   workerpermC[1]    := FALSE;

   workerpermC[2]    := FALSE;
  workerpermC[3]    := FALSE;


  workerpermD[0]    := TRUE;

  workerpermD[1]    := TRUE;

  workerpermD[2]    := FALSE;

  workerpermD[3]    := FALSE;


-- Minimum Duties

SPEC AG( (sectDpermA[2] & !sectDpermA[3]) | (sectDpermB[2] &!sectDpermB[3]) |
(sectDpermC[2] & !sectDpermC[3]) | (sectDpermD[2] & !sectDpermD[3]))


-- Static Separation of Duties

SPEC AG(( (sectDpermA[2]) -> AG (!sectDpermB[2] & !sectDpermC[2] & !sectDpermD[2]))
& ((sectDpermB[2]) -> AG (!sectDpermA[2] & !sectDpermC[2] & !sectDpermD[2])) &
((sectDpermC[2]) -> AG (!sectDpermA[2] & !sectDpermB[2] & !sectDpermD[2])) &
((sectDpermD[2]) -> AG (!sectDpermA[2] & !sectDpermB[2] & !sectDpermC[2]) ))


--Role Hierarchy

SPEC AG( !(sectDpermA[0]) -> AG !(workerpermA[0]) )

SPEC AG( (sectDpermA[0] & !sectDpermA[1]) -> AG !(workerpermA[0]) )

SPEC AG( (sectDpermA[1] & !sectDpermA[2]) -> AG !(workerpermA[1]) )

SPEC AG( (sectDpermA[2] & !sectDpermA[3]) -> AG !(workerpermA[2]) )

SPEC AG( (sectDpermA[3]) -> AG !(workerpermA[3]))


SPEC AG( !(sectDpermB[0]) -> AG !(workerpermB[0]) )
SPEC AG( (sectDpermB[0] & !sectDpermB[1]) -> AG !(workerpermB[0]) )

SPEC AG( (sectDpermB[1] & !sectDpermB[2]) -> AG !(workerpermB[1]) )

SPEC AG( (sectDpermB[2] & !sectDpermB[3]) -> AG !(workerpermB[2]) )

SPEC AG( (sectDpermB[3]) -> AG !(workerpermB[3]))



SPEC AG( !(sectDpermC[0]) -> AG !(workerpermC[0]) )

SPEC AG( (sectDpermC[0] & !sectDpermC[1]) -> AG !(workerpermC[0]) )

SPEC AG( (sectDpermC[1] & !sectDpermC[2]) -> AG !(workerpermC[1]) )

SPEC AG( (sectDpermC[2] & !sectDpermC[3]) -> AG !(workerpermC[2]) )

SPEC AG( (sectDpermC[3]) -> AG !(workerpermC[3]))



SPEC AG( !(sectDpermD[0]) -> AG !(workerpermD[0]) )

SPEC AG( (sectDpermD[0] & !sectDpermD[1]) -> AG !(workerpermD[0]) )

SPEC AG( (sectDpermD[1] & !sectDpermD[2]) -> AG !(workerpermD[1]) )

SPEC AG( (sectDpermD[2] & !sectDpermD[3]) -> AG !(workerpermD[2]) )

SPEC AG( (sectDpermD[3]) -> AG !(workerpermD[3]))



----------------- End of sectionLeader D



MODULE workerA(projectA, projectB, projectC, projectD, workerpermA,
workerpermB,workerpermC, workerpermD)

VAR

   userWorkerA : process User(projectA, projectB, projectC, projectD, workerpermA,
workerpermB, workerpermC, workerpermD);
```

```
-- Minimum Duties

SPEC AG( (workerpermA[1] & !workerpermA[2]) | (workerpermB[1] &!workerpermB[2]) |
(workerpermC[1] & !workerpermC[2]) | (workerpermD[1] & !workerpermD[2]))



-- Static Separation of Duties

SPEC AG(( (workerpermA[1]) -> AG (!workerpermB[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermB[1]) -> AG (!workerpermA[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermC[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermD[1])) & ((workerpermD[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermC[1]) ))



MODULE workerB(projectA, projectB, projectC, projectD, workerpermA,
workerpermB,workerpermC, workerpermD)

VAR

   userWorkerB : process User(projectA, projectB, projectC, projectD, workerpermA,
workerpermB, workerpermC, workerpermD);



-- Minimum Duties

SPEC AG( (workerpermA[1] & !workerpermA[2]) | (workerpermB[1] &!workerpermB[2]) |
(workerpermC[1] & !workerpermC[2]) | (workerpermD[1] & !workerpermD[2]))



-- Static Separation of Duties

SPEC AG(( (workerpermA[1]) -> AG (!workerpermB[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermB[1]) -> AG (!workerpermA[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermC[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermD[1])) & ((workerpermD[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermC[1]) ))
```

```
MODULE workerC(projectA, projectB, projectC, projectD, workerpermA,
workerpermB,workerpermC, workerpermD)

VAR

  userWorkerC : process User(projectA, projectB, projectC, projectD, workerpermA,
workerpermB, workerpermC, workerpermD);



-- Minimum Duties

SPEC AG( (workerpermA[1] & !workerpermA[2]) | (workerpermB[1] &!workerpermB[2]) |
(workerpermC[1] & !workerpermC[2]) | (workerpermD[1] & !workerpermD[2]))



-- Static Separation of Duties

SPEC AG(( (workerpermA[1]) -> AG (!workerpermB[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermB[1]) -> AG (!workerpermA[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermC[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermD[1])) & ((workerpermD[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermC[1]) ))



MODULE workerD(projectA, projectB, projectC, projectD, workerpermA,
workerpermB,workerpermC, workerpermD)

VAR

  userWorker : process User(projectA, projectB, projectC, projectD, workerpermA,
workerpermB, workerpermC, workerpermD);



-- Minimum Duties

SPEC AG( (workerpermA[1] & !workerpermA[2]) | (workerpermB[1] &!workerpermB[2]) |
(workerpermC[1] & !workerpermC[2]) | (workerpermD[1] & !workerpermD[2]))



-- Static Separation of Duties
```

```
SPEC AG(( (workerpermA[1]) -> AG (!workerpermB[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermB[1]) -> AG (!workerpermA[1] & !workerpermC[1] &
!workerpermD[1])) & ((workerpermC[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermD[1])) & ((workerpermD[1]) -> AG (!workerpermA[1] & !workerpermB[1] &
!workerpermC[1]) ))


MODULE User(projectA, projectB, projectC, projectD, permA, permB, permC, permD)

VAR


  myCommandA : { Wait, Read, Write, Document, Schedule};

  myCommandB : { Wait, Read, Write, Document, Schedule};

  myCommandC : { Wait, Read, Write, Document, Schedule};

  myCommandD : { Wait, Read, Write, Document, Schedule};


ASSIGN


  init(myCommandA) := Wait;

  init(myCommandB) := Wait;

  init(myCommandC) := Wait;

  init(myCommandD) := Wait;


  next(myCommandA) := case


                          (permA[0] = TRUE) & (permA[1] = FALSE) & (permA[2] = FALSE) &
(permA[3] = FALSE) : {Wait,Read};
```

```
                  (permA[0] = TRUE) & (permA[1] = TRUE) & (permA[2] = FALSE) &
(permA[3] = FALSE)  : {Wait,Read, Write};

                  (permA[0] = TRUE) & (permA[1] = TRUE) & (permA[2] = TRUE) &
(permA[3] = FALSE)  : {Wait,Read, Write,Document};

                  (permA[0] = TRUE) & (permA[1] = TRUE) & (permA[2] = TRUE) &
(permA[3] = TRUE) : {Wait,Read, Write,Document, Schedule};

                          TRUE   :  Wait;

                          esac;



 next(projectA.state) :=

                         case

                         myCommandA != Wait : myCommandA;

                         TRUE   : Wait;

                         esac;




 next(myCommandB) := case



                  (permB[0] = TRUE) & (permB[1] = FALSE) & (permB[2] = FALSE) &
(permB[3] = FALSE) : {Wait,Read};

                  (permB[0] = TRUE) & (permB[1] = TRUE) & (permB[2] = FALSE) &
(permB[3] = FALSE) : {Wait,Read, Write};

                  (permB[0] = TRUE) & (permB[1] = TRUE) & (permB[2] = TRUE) &
(permB[3] = FALSE) : {Wait,Read, Write, Document};

                  (permB[0] = TRUE) & (permB[1] = TRUE) & (permB[2] = TRUE) &
(permB[3] = TRUE) : {Wait,Read, Write, Document, Schedule};
```

```
                              TRUE   :  Wait;

                              esac;


 next(projectB.state) :=

                            case

                            myCommandB != Wait : myCommandB;

                            TRUE   : Wait;

                            esac;




 next(myCommandC) := case


                        (permC[0] = TRUE) & (permC[1] = FALSE) & (permC[2] = FALSE) &
(permC[3] = FALSE) : {Wait,Read};

                        (permC[0] = TRUE) & (permC[1] = TRUE) & (permC[2] = FALSE) &
(permC[3] = FALSE) : {Wait,Read, Write};

                        (permC[0] = TRUE) & (permC[1] = TRUE) & (permC[2] = TRUE) &
(permC[3] = FALSE) : {Wait,Read, Write, Document};

                        (permC[0] = TRUE) & (permC[1] = TRUE) & (permC[2] = TRUE) &
(permC[3] = TRUE) : {Wait,Read, Write, Document, Schedule};

                              TRUE   :  Wait;

                              esac;


 next(projectC.state) :=
                        case

                        myCommandC != Wait : myCommandC;

                        TRUE   : Wait;

                        esac;



  next(myCommandD) := case



                  (permD[0] = TRUE) & (permD[1] = FALSE) & (permD[2] = FALSE) &
(permD[3] = FALSE) : {Wait,Read};

                  (permD[0] = TRUE) & (permD[1] = TRUE) & (permD[2] = FALSE) &
(permD[3] = FALSE) : {Wait,Read, Write};

                  (permD[0] = TRUE) & (permD[1] = TRUE) & (permD[2] = TRUE) &
(permD[3] = FALSE) : {Wait,Read, Write, Document};

                  (permD[0] = TRUE) & (permD[1] = TRUE) & (permD[2] = TRUE) &
(permD[3] = TRUE) : {Wait,Read, Write, Document, Schedule};

                         TRUE   :  Wait;

                         esac;



  next(projectD.state) :=

                        case

                        myCommandD != Wait : myCommandD;

                        TRUE   : Wait;

                        esac;
```

```
-- Standard Access Control Checks.  If you don't have the permission, you can't do the command.

SPEC AG ! (permA[0] = FALSE & myCommandA = Read)

SPEC AG ! (permA[1] = FALSE & myCommandA = Write)

SPEC AG ! (permA[2] = FALSE & myCommandA = Document)

SPEC AG ! (permA[3] = FALSE & myCommandA = Schedule)



SPEC AG ! (permB[0] = FALSE & myCommandB = Read)

SPEC AG ! (permB[1] = FALSE & myCommandB = Write)

SPEC AG ! (permB[2] = FALSE & myCommandB = Document)

SPEC AG ! (permB[3] = FALSE & myCommandB = Schedule)



SPEC AG ! (permC[0] = FALSE & myCommandC = Read)

SPEC AG ! (permC[1] = FALSE & myCommandC = Write)

SPEC AG ! (permC[2] = FALSE & myCommandC = Document)

SPEC AG ! (permC[3] = FALSE & myCommandC = Schedule)



SPEC AG ! (permD[0] = FALSE & myCommandD = Read)

SPEC AG ! (permD[1] = FALSE & myCommandD = Write)

SPEC AG ! (permD[2] = FALSE & myCommandD = Document)

SPEC AG ! (permD[3] = FALSE & myCommandD = Schedule)
```

```
-- Sets hierachy of the permissions.  If you have a higher permission, you should have the lower
too.

SPEC AG (  (permA[1]) -> (permA[0]) )

SPEC AG (  (permA[2]) -> (permA[0] & permA[1]) )

SPEC AG (  (permA[3]) -> (permA[0] & permA[1] & permA[2]) )



SPEC AG (  (permB[1]) -> (permB[0]) )

SPEC AG (  (permB[2]) -> (permB[0] & permB[1]) )

SPEC AG (  (permB[3]) -> (permB[0] & permB[1] & permB[2]) )



SPEC AG (  (permC[1]) -> (permC[0]) )

SPEC AG (  (permC[2]) -> (permC[0] & permC[1]) )

SPEC AG (  (permC[3]) -> (permC[0] & permC[1] & permC[2]) )



SPEC AG (  (permD[1]) -> (permD[0]) )

SPEC AG (  (permD[2]) -> (permD[0] & permD[1]) )

SPEC AG (  (permD[3]) -> (permD[0] & permD[1] & permD[2]) )



MODULE userSession(semLA, semLB, semLC, semLD, semS, givenRoles, myName)

VAR
```

```
   activeRole : {loggedOut, workerA, workerB, workerC, workerD, leaderA, leaderB, leaderC,
leaderD, supervisor};

ASSIGN

   init(activeRole) := {loggedOut};

   next(activeRole) := case
                              (givenRoles[8] & !semS.sema) : {supervisor};
                              (givenRoles[7] & !semLD.sema) : {leaderD};
                              (givenRoles[6] & !semLC.sema) : {leaderC};
                              (givenRoles[5] & !semLB.sema) : {leaderB};
                              (givenRoles[4] & !semLA.sema) : {leaderA};
                              (givenRoles[3]) : {workerD};
                              (givenRoles[2]) : {workerC};
                              (givenRoles[1]) : {workerB};
                              (givenRoles[0]) : {workerA};
                              TRUE : loggedOut;
                              esac;

   next(semS.sema) := case
                        activeRole = supervisor : TRUE;
```

```
                    activeRole != supervisor & semS.userName = myName : FALSE;
                    TRUE : FALSE;
                  esac;

next(semS.userName) := case
                    activeRole = supervisor : myName;
                    TRUE : None;
                  esac;

next(semLD.sema) := case
                    activeRole = leaderD : TRUE;
                    activeRole != leaderD & semLD.userName = myName : FALSE;
                    TRUE : FALSE;
                  esac;

next(semLD.userName) := case
                    activeRole = leaderD : myName;
                    TRUE : None;
                  esac;
```

```
next(semLC.sema) := case
            activeRole = leaderC : TRUE;
            activeRole != leaderC & semLC.userName = myName : FALSE;
            TRUE : FALSE;
           esac;



next(semLC.userName) := case
            activeRole = leaderC : myName;
            TRUE : None;
           esac;

next(semLB.sema) := case
            activeRole = leaderB : TRUE;
            activeRole != leaderB & semLB.userName = myName : FALSE;
            TRUE : FALSE;
           esac;

next(semLB.userName) := case
            activeRole = leaderB : myName;
            TRUE : None;
           esac;
```

```
  next(semLA.sema) := case

                       activeRole = leaderA : TRUE;

                       activeRole != leaderA & semLA.userName = myName : FALSE;

                       TRUE : FALSE;

                     esac;



  next(semLA.userName) := case

                       activeRole = leaderA : myName;

                       TRUE : None;

                     esac;

-- DSOD for Worker Roles

SPEC AG ( givenRoles[0] -> AG !(givenRoles[1] | givenRoles[2]| givenRoles[3] | givenRoles[5]
| givenRoles[6] | givenRoles[7]) | (givenRoles[8])  )

SPEC AG ( givenRoles[1] -> AG !(givenRoles[0] | givenRoles[2]| givenRoles[3] | givenRoles[4]
| givenRoles[6] | givenRoles[7]) | (givenRoles[8])  )

SPEC AG ( givenRoles[2] -> AG !(givenRoles[0] | givenRoles[1]| givenRoles[3] | givenRoles[4]
| givenRoles[5] | givenRoles[7]) | (givenRoles[8])  )

SPEC AG ( givenRoles[3] -> AG !(givenRoles[0] | givenRoles[1]| givenRoles[2] | givenRoles[4]
| givenRoles[6] | givenRoles[6]) | (givenRoles[8])  )



-- DSOD for Leader Roles

SPEC AG ( givenRoles[4] -> AG !(givenRoles[1] | givenRoles[2] | givenRoles[3] |
givenRoles[5] | givenRoles[6] | givenRoles[7]) | (givenRoles[8]) )

SPEC AG ( givenRoles[5] -> AG !(givenRoles[0] | givenRoles[2] | givenRoles[7] |
givenRoles[4] | givenRoles[6] | givenRoles[7]) | (givenRoles[8]) )
```

```
SPEC AG ( givenRoles[6] -> AG !(givenRoles[0] | givenRoles[1] | givenRoles[3] |
givenRoles[4] | givenRoles[5] | givenRoles[7]) | (givenRoles[8]) )

SPEC AG ( givenRoles[7] -> AG !(givenRoles[0] | givenRoles[1] | givenRoles[2] |
givenRoles[4] | givenRoles[5] | givenRoles[6]) | (givenRoles[8]) )
```