4-2014

The Application of Graph Theory to Access Control Systems

Eric Brown
University of Nebraska at Omaha

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Part of the Computer Sciences Commons

Please take our feedback survey at: https://unomaha.az1.qualtrics.com/jfe/form/SV_8cchtFmpDyGfBLE

Recommended Citation

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
The Application of Graph Theory to Access Control Systems

A Thesis
Presented to the
Department of Computer Science
and the
Faculty of the Graduate College
University of Nebraska
in Partial Fulfillment
of the Requirements for the Degree
Master of Science
University of Nebraska at Omaha
by
Eric Brown
April, 2014

Supervisory Committee
Victor Winter, Ph. D.
Harvey Siy, Ph. D.
William Mahoney, Ph. D.
Abstract

Computer systems contain vital information that must be protected. One of the crucial aspects of protection is access control. A review of some of the research into ways in which access to the information in computers can be controlled focuses on a question about safety. The safety question asks, “Can a user ever gain access to a resource for which he is not authorized?” This question cannot be answered in general because of the unbounded, unrestricted nature of a general-purpose access control system. It can be answered only for systems that are specifically designed to restrict the actions that can be taken by users of a system so that all of its possible future states, with respect to a certain right ‘leaking’ to an unauthorized user, can be predicted. There is a tension between the power to take useful actions within a system, and the ability to predict safety.
Another way to ensure safety is to express all of a system’s security policies as constraints. Then the access control system itself can be unrestricted, but each action is checked to ensure that no security policy has been violated so far. These are very complex and difficult to manage and, therefore, are difficult systems for which to verify the safety property.

This thesis models an alternative access control system, consolidating disparate research and balancing expressive power and safety analysis. The system, called Graph Plus, uses a graph representation of the protection state. It has decidable safety algorithms, and many useful operations. The usefulness is further enhanced by combining the constraint philosophy with the restricted model philosophy into one hybrid system. Additional operations have been included that can be analyzed by the efficient safety algorithms of the underlying model. This safety analysis acts as a constraint upon the additional operations, so that safety can be retained even with the additional operations added to the model. The model and its implementation are described.
Dedication

This thesis is dedicated to the Lord, Jesus Christ. Without him, my next breath would be impossible, not to mention a long-lasting endeavor such as graduate school. May He receive all of the praise.
Acknowledgements

I would like to thank my wife, Joan, and all of my children for their patience during the thesis process and for all of the years of graduate school. Thank you for your support and encouragement.

I would also like to thank Dr. Victor Winter and the other members of my Thesis committee: Dr. Harvey Siy and Dr. Bill Mahoney. Thank you for your patience during two full years of long delays punctuated by flurries of activity.
# Table of Contents

The Application of Graph Theory to Access Control Systems ........................................... i

The Application of Graph Theory to Access Control Systems ........................................... ii

Abstract.................................................................................................................................. ii

Dedication ................................................................................................................................. iv

Acknowledgements.................................................................................................................. v

List of Figures ......................................................................................................................... viii

1 Introduction .......................................................................................................................... 1

1.1 Organization of the Thesis ............................................................................................... 7

2 Related Work ....................................................................................................................... 8

2.1 Harrison, Ruzzo and Ullman ........................................................................................... 8

2.2 Safety ............................................................................................................................... 15

2.3 Sandhu ............................................................................................................................. 19

2.4 Koch ................................................................................................................................. 19

2.5 Lipton and Snyder ............................................................................................................ 21

2.6 Lockman and Minsky ....................................................................................................... 26

2.7 Constraint Based Safety ................................................................................................... 29

2.8 Next Steps ....................................................................................................................... 30

3 An Alternative .................................................................................................................... 31

4 The Model .......................................................................................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Representation</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Operations</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Weak Edges</td>
<td>46</td>
</tr>
<tr>
<td>4.4</td>
<td>Safety</td>
<td>50</td>
</tr>
<tr>
<td>4.5</td>
<td>Usefulness</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>A Useful Illustration</td>
<td>67</td>
</tr>
<tr>
<td>5.1</td>
<td>The Model’s Representation</td>
<td>68</td>
</tr>
<tr>
<td>5.2</td>
<td>Adding Rights</td>
<td>68</td>
</tr>
<tr>
<td>5.3</td>
<td>Closure</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Visualization</td>
<td>72</td>
</tr>
<tr>
<td>5.5</td>
<td>Persistence</td>
<td>72</td>
</tr>
<tr>
<td>6</td>
<td>Conclusions</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>Future Work</td>
<td>75</td>
</tr>
<tr>
<td>6.2</td>
<td>References</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td>Appendix</td>
<td>78</td>
</tr>
<tr>
<td>7</td>
<td>Appendix A – Proofs of Safety Claims</td>
<td>79</td>
</tr>
<tr>
<td>8</td>
<td>Appendix B – graphPlus Source Code</td>
<td>86</td>
</tr>
<tr>
<td>8.1</td>
<td>AdminLib.java</td>
<td>86</td>
</tr>
<tr>
<td>8.2</td>
<td>GraphAdapter.java</td>
<td>90</td>
</tr>
<tr>
<td>8.3</td>
<td>GraphDisplay.java</td>
<td>95</td>
</tr>
</tbody>
</table>
8.4 GraphMain.java ................................................................. 99
8.5 GraphModel.java ............................................................. 101
8.6 ListenableDirectedWeightedMultiGraph.java ....................... 108
8.7 PersistentEdge.java ......................................................... 108
8.8 Right.java .................................................................. 109
8.9 SimpleLibrary.java ......................................................... 111
8.10 TransitiveClosureMulti.java ............................................ 116
8.11 Utils.java .................................................................. 117
8.12 Vertex.java .................................................................. 119

List of Figures

Figure 2-1 ............................................................................. 22
Figure 2-2 ............................................................................. 22
Figure 2-3 ............................................................................. 23
Figure 2-4 ............................................................................. 23
Figure 2-5 ............................................................................. 24
Figure 2-6 ............................................................................. 26
Figure 3-1 ............................................................................. 32
Figure 3-2 ............................................................................. 33
Figure 3-3 ................................................................................................................. 33
Figure 3-4 ................................................................................................................. 34
Figure 4-1 ..................................................................................................................... 39
Figure 4-2 ..................................................................................................................... 41
Figure 4-3 ..................................................................................................................... 44
Figure 4-4 ..................................................................................................................... 45
Figure 4-5 ..................................................................................................................... 47
Figure 4-6 ..................................................................................................................... 48
Figure 4-7 ..................................................................................................................... 49
Figure 4-8 ..................................................................................................................... 53
Figure 4-9 ..................................................................................................................... 54
Figure 4-10 .................................................................................................................. 60
Figure 4-11 ................................................................................................................. 61
Figure 4-12 .................................................................................................................. 62
Figure 4-13 .................................................................................................................. 63
Figure 4-14 .................................................................................................................. 65
Figure 4-15 .................................................................................................................. 66
Figure 4-16 .................................................................................................................. 66
Figure 5-1 ..................................................................................................................... 69
Figure 5-2 ..................................................................................................................... 71
Figure 7-1 ..................................................................................................................... 83
1 Introduction

Today’s computer systems and their interconnections with other systems have a larger capacity to store information and provide services than ever before. As computer systems have become more widely used, businesses, institutions and individuals have come to rely upon the information they store and the services they provide. This dependence has resulted in the realization that the loss, corruption, or even interruption of the flow of data on demand can have detrimental and even catastrophic consequences.

Information, regardless of the media upon which it is stored, has always been subject to loss, unauthorized disclosure, and a range of other threats. It has therefore always needed protection. Consider the banking industry and its duty to keep the financial records of clients secret, or a military installation safeguarding classified information. The information in a computer system must also be guarded. Many of the same threats that are common to non-computerized information are also faced by a computer system, [1] and sometimes those threats are amplified by the nature of computer systems. Which people should have access to the information? How can you tell if someone is who he claims to be? Who should be allowed to make copies? All of these
questions and many more are faced by computer systems and non-computerized information stores alike.

Computers also face new kinds of threats, such as computer viruses and other kinds of malicious software, network attacks in which a computer system is targeted for some kind of scan, or perhaps a specially designed service request that exploits a vulnerability somewhere in the system.

Protecting the information and services that a computer system provides is known as Computer Security. As our reliance on computer systems grows, so will our interest in Computer Security.

One critical aspect of Computer Security is the ability to control which user can gain access to which resources. Access Control in a non-computerized information store is accomplished by means of physical barriers such as safes and locked doors, guarded turnstiles and identity verification. In a computer system there must also be a method to control access to the information and resources within that particular system. An access control mechanism for a computer system must allow users to gain access to the information resources they need, and for which they are authorized. It should also allow users to give access to the information they control to other users, while enforcing policies that meet the goals of the organization. For example, the members of a
company’s Human Resources department might be allowed to give access to employee files, which they control, to other members of the Human Resources department, but not to employees that are not part of Human Resources. The part of a computer system that controls the sharing of information between users is called the access control system or the protection system.

There must be some way for the access control system to keep track of the resources being protected, the users of the system, and what kind of access each user is allowed to each resource. This combination of users, their access privileges, and the resources being protected is called the access control system’s state, or the protection state.

Controlling the way in which the information and services within computer systems can be shared, because of its critical nature, has been the subject of much research. Some of the research has focused on the question of safety, which asks if the access control system can be manipulated by any sequence of events in such a way that a user can gain access to resources for which he is not truly authorized. Using a given access control system, would it be possible for some unauthorized user to ever gain access to such resources? What kind of access could be gained? Can it be proven that unauthorized access can never occur? These kinds of questions culminate in what is known as the safety question: can the system ever reach a state in which a particular user possesses a
particular privilege for a specific resource? The answer to this question has been elusive.

Some systems have very few limits on the operations that are available to make changes to the \textit{protection state}. These systems allow users to take many kinds of actions such as creating new users, granting other users the ability to read and write files, copying files, etc. Collectively, the rules governing the actions that can be taken in a system are called \textit{security policies}. Examples of security policies include:

- Any user can create a new user
- Any user can create a new file
- The creator of a file has permission to read, write, and execute that file.

A flexible system that allows many different security policies is said to be \textit{expressive} or to have \textit{expressive power}. When a system is more restrictive, carefully controlling the way in which users can alter the protection state, it is said to be less expressive. In general, there is an inverse relationship between the expressive power of a protection system and its capability to predict if access to an object or resource can be gained by an unauthorized user. For many systems, it cannot be predicted at all. Safety for these systems is called \textit{undecidable}.

On the other hand, if the system is very restrictive and bounded, so that users cannot make as many changes, then the safety of the system is predictable. This balance
between safety and expressiveness is at the core of protection system design, as Sandhu points out [2]:

There is an essential conflict between the expressive power of an access control model and tractability of safety analysis.

Over the decades since computing became widespread, there have been different approaches to representing the state of an access control system; that is, which users have access to which resources. There have been many models, restrictions, algorithms, and design ideas. [3] [4] [1] [5] [6]

Some of these models use a matrix of rows and columns to represent the users and files in a system, and the rights belonging to those users. Other suggestions include representing the state of the access control system using a graph of vertices and interconnecting edges.

Many of these proposed systems provide proofs that, if the operations on the access control model are limited in some way, the safety question with respect to a given right can be answered, perhaps efficiently. However, in order to obtain these results, these models have had to limit the operations they can include, sometimes trading the assurance of safety for the ability to include any useful security policies, such as the Originator Control (ORCON) policy described in Table 1-1.
<table>
<thead>
<tr>
<th>Rule</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Users can create files</td>
<td>CreateFile – establishes a new file and records the user that created it as the owner</td>
</tr>
<tr>
<td>Access to a file can only be obtained from the file’s owner</td>
<td>Grant – gives another user a specific access to a file</td>
</tr>
<tr>
<td>Access to a file can be revoked by the file’s owner</td>
<td>Revoke – takes away a specific access from another user</td>
</tr>
</tbody>
</table>

Table 1-1

Other proposals shift the focus from the way in which the state of the access control system is represented, to the way in which safety is verified. These schemes allow the access control system to be very flexible and expressive; however, when some operation is invoked in the access control system, then all of the policies of the organization are checked to make sure that the operation did not violate any of them. Instead of trying to build a system to prevent safety violations, they constrain each operation as it is being run to verify that no violation occurs. These tend to be very complex and difficult for users to manage [4].

The problem is that the best ideas are fragmented among the papers and articles produced over the past three decades. There seems to be an opportunity to consolidate these findings [7], and make progress in applying graph theory results to the problem of safety in access control.
This thesis will present an alternative approach that combines the use of a graph-based representation of the state of the access control system and run time constraints to monitor the safety of the system at run time in a hybrid system. The hybrid system will be modeled, explained, implemented, and demonstrated. The conclusions and insights gained by the implementation and use of the hybrid system will be considered, along with opportunities for future work.

1.1 Organization of the Thesis

The remaining part of the Thesis is organized as follows:

- Chapter 2 is a review of the important literature that bears upon this thesis. The development of the various Access Control mechanisms and their contribution to the safety question will be reviewed.

- Chapter 3 is the proposal of an alternative system design for access control using a hybrid approach, which is the main contribution of this Thesis.

- Chapter 4 is the delineation of the theoretical model for the new system, how it extends and is supported by previous related work, and how it includes useful security policies.

- Chapter 5 is presents the research implementation of the new system and its representation and algorithms.
Chapter 6 describes the insights that have been gained by this research, suggestions for further study, and the conclusion of the Thesis.

2 Related Work

There have been a number of research papers written on the subject of safety in access control. In the following sections, I review the papers that have made a contribution important to the development of this Thesis.

It will be useful to keep some concepts in mind that are common to all of the access control models discussed in this chapter. Each model defines a set of Subjects, a set of Objects, a set of Access Rights, and a set of Operations. The subjects are the active entities in a system, such as users and processes. Objects are those entities that subjects act upon, such as files and even other Subjects. Rights define the kind of access that a subject is allowed over an object, such as Read, Write, Execute, Delete, Append, etc. Operations are the actions that the subjects can take upon the objects, such as Read, Grant, create, and remove.

2.1 Harrison, Ruzzo and Ullman

The classic formalization of access control in computer systems is the well-known HRU model [6], so-called because of its authors: Harrison, Ruzzo, and Ullman. In this model, the protection state is represented by an access control matrix. A matrix can be thought
of as a table, like a multiplication table, with rows and columns. The access control matrix represents the state of the access control system: that conglomeration of all of the Subjects, Objects and their access rights. In the access control matrix, there is a row for each subject, and a column for each object. In this model, Subjects can also be considered objects, so each subject has a column in addition to its row. The intersection of row X and column Y is referred to as the \([X,Y]\) cell. The value in the \([X,Y]\) cell contains the access rights the subject X has to the object Y. Table 2-1 is an example of an access control matrix.

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RWE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RW</td>
<td></td>
</tr>
<tr>
<td>P4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RW</td>
</tr>
</tbody>
</table>

Table 2-1

The operations in the HRU model are called commands, and each command must be of the form:

\[
\text{command-name}(X_1, X_2, \ldots, X_3)
\]
\[
\text{if } r_1 \text{ in } (X_{s1}, X_{o1}) \text{ and}
\]
\[
\quad r_2 \text{ in } (X_{s2}, X_{o2}) \text{ and}
\]
\[
\quad \ldots
\]
\[
\quad r_m \text{ in } (X_{sm}, X_{om})
\]
\[
\text{then}
\]
\[
\text{op}_1
\]
where $X_1 \ldots X_k$ are formal parameters, $r_1 \ldots r_m$ are generic rights, $s_1, \ldots, s_m$ and $o_1, \ldots, o_m$ are integers between 1 and $k$. The first section, from the ‘if’ to the ‘then’ is called the condition, and may be omitted. The section after the ‘then’ is called the body.

The condition checks to see if the rights required by the command are present in the matrix. If so, the body is executed. The body contains one or more simple operations called primitive operations, or primitives. The primitive operations are:

- Enter $r$ into $(X, Y)$ – enters one of the previously defined rights into the matrix at the $[X,Y]$ cell.
- Delete $r$ from $(X, Y)$ – deletes one of the previously defined rights from the matrix at the $[X,Y]$ cell.
- Create subject $(X)$ – Creates a row (and a column) for this subject in the matrix
- Create object $(Y)$ – Creates a column for this object in the matrix
- Destroy subject $(X)$ – Deletes the row and Column for this subject from the matrix
- Destroy object $(Y)$ – Deletes the column for this object from the matrix.

The formal semantics of such a model are quite important as they quantify exactly how the state of the access control system must change for each operation. In HRU, the following definitions formally define the model:
• $R = \{r, w, x, o\}$ is a set of rights defined for this system. They symbolize the read, write, execute, and own rights, respectively.

• The protection state is an ordered triple $\{S, O, P\}$ where $S$ is the set of current subjects, $O$ is the set of current objects, $S \subseteq O$, and $P$ is an access matrix with a row for every subject in $S$, and a column for every object in $O$.

• $P[s, o]$ denotes the contents of the cell at the intersection of the row for subject $s$ and the column for object $o$, and $P[s, o] \subseteq R$.

• Let $\{S, O, P\}$ and $\{S', O', P'\}$ be instances of the protection state.

• $\{S, O, P\} \Rightarrow_{op} \{S', O', P'\}$ (read: $\{S, O, P\}$ yields $\{S', O', P'\}$ under op) if op is one of the following:
  
  o enter $r$ into $(s, o)$ and
    - $S = S'$, $O = O'$, $s \in S$, $o \in O$,
    - $P'[s_1, o_1] = P[s_1, o_1] \forall [s_1, o_1] \neq [s, o]$
    - $P'[s, o] = P[s, o] \cup \{r\} $
  
  o delete $r$ from $(s, o)$ and
    - $S = S'$, $O = O'$, $s \in S$, $o \in O$,
    - $P'[s_1, o_1] = P[s_1, o_1] \forall [s_1, o_1] \neq [s, o]$
    - $P'[s, o] = P[s, o] \setminus \{r\}$
  
  o create subject $s'$ where $s' \notin O$, $S' = S \cup \{s'\}$, $O' = O \cup \{s'\}$
    - $P'[s, o] = P[s, o] \forall (s, o) \in S \times O$ and
    - $P'[s', o] = \emptyset \forall o \in O'$
create object o’ where o’ ∉ O, S’ = S, O’ = O U {o’} and

- P’[s, o] = P[s, o] ∀ (s, o) in S x O
- P’[s, o’] = φ ∀ s ∈ S’

destroy subject s’, where s’ ∈ S, S’ = S – {s’}, O’ = O – {s’} and

- P'[s,o] = P[s,o] ∀ (s,o) in S’ x O’

destroy object o’, where o’ ∈ O – S, S’ = S, O’ = O – {o’} and

- P'[s,o] = P[s,o] ∀ (s,o) in S’ x O’

The subjects in the HRU protection system cannot run these six primitives directly; they can only invoke commands, which determine if the matrix contains the necessary rights for the operations to be performed. Collectively, these commands constitute the overall security policy for the system. The commands must be defined at the time of the system’s initial configuration, as are the set of rights that the system contains. However, the commands must have the defined structure and a (possibly empty) set of conditions. The authors give three commands as an example:

- **Create**(subject, file){

  create object file;

  enter own into (subject, file);

}

- **Confer**(r, owner, friend, file){

  If (own in P[owner, file]){


Enter $r$ into $(\text{friend}, \text{file})$;

$\}$

$\}$$

- Remove($r$, owner, exfriend, file){
  
  If own in P[owner, file]{
    
    delete $r$ from (exfriend, file);
  
  }

}

Using these three commands and the set of rights $R = \{r,w,x,o\}$, the security policies that this instance of the model can express are:

- Users can create files, which they own;
- Users can grant other users the rights to access files which they own.
- Users can revoke from other users the rights to access files which they own.

Collectively, these policies implement the ORCON security policy from Table 1-1.

If the system administrator designs more complex commands, with a different set of rights, then the system can express many more security policies. Depending on which rights are used, and which commands are designed, this protection system can model just about any security policy one can imagine.
Once the initial State has been established, it can be changed by the autonomous action of the subjects invoking the operations inherent in the model upon the system.

To add a new user, an AddUser command could be built from the primitive operations that the model supplies. A new right could be added, perhaps ‘a’ for ‘administrator’. When the AddUser command is invoked, it will verify the presence of the ‘a’ right in the subject that invokes it, then add a new row and column to the state of the protection system for the new user. If the new user needed a right on some other object, the Confer command would be invoked to grant that right on that object to the specified user. Operations such as these alter the state of the protection system either by inserting new rows/columns, or by adding rights into existing cells.

For HRU, since any commands and rights can be configured in the system (i.e. they are not restricted), the protections state can grow (by adding new users) and change (in ways that are not restricted by the model) in an unbounded way. This makes the model very expressive of security policies, but also makes the safety question undecidable, as outlined in the next section.

In our example, there exist not only those rights which give the subject certain kinds of access to the object, such as read, write and execute, but there is also the ‘own’ right. The ‘owner’ of a system object has the right to confer upon any other subject any of the
rights that the owner has over that object (except ‘own’). This right controls the transfer of other rights, and therefore becomes a most interesting part of any analysis of safety – which is concerned with allowing no rights to ‘leak’ to an unauthorized subject. Rights such as ‘own’ in this instance of HRU, commonly known as the transport rights in subsequent literature, form an interesting class of rights. The other rights can be conferred and removed, but they do not play any role in controlling which subjects can invoke those commands. Transport rights restrict the flow of rights from one subject to another in the access control matrix. As such, the transport rights suggest an avenue by which the issue of safety might be addressed.

This HRU access control matrix has been the basis for a majority of the subsequent research in the protection of computer systems. In addition, many of the access control implementations in use by actual computer systems correspond to this model. [1]

2.2 Safety

Designers of these Access Control Systems must consider the cumulative effect of the incremental changes to the state of the system in order to understand what states are reachable by any sequence of subjects invoking operations on the system. Would it be possible for a subject to obtain a right over an object that it was not intended to have? Further, can an algorithm be run to decide whether a protection system in a certain state could ever reach state in which a subject had a right that it was not intended to
have? The ultimate question that arises, called the safety question, is stated well by Sandhu in [2]:

“...the safety question asks: is there a reachable state in which a particular subject possesses a particular privilege for a specific object? It is the fundamental question which an access control model must confront. Since subjects are usually authorized to create new subjects and objects, the system is unbounded; and it is not certain that such analysis will be decidable, let alone tractable, without sacrificing generality.”

Intuitively, since the state can be changed in a way that is unbounded, and there are few restrictions on its expressive power, there can be no algorithm by which we can prove that a given change to the system will not result in a right “leaking” to a subject for whom it was not intended. This has actually been demonstrated by [6], in which Harrison et al reduce the problem to the famous Halting problem of Alan Turing [8].

Turing reasoned that if it were possible to specify a sequence of operations, which a machine could interpret and perform, and then to reach a final state and halt, then that sequence of instructions taken as a whole could be considered a ‘computable function.’ We would call it an algorithm. He realized not only that there are some problems that are not computable by any sequence of operations, but that, in general, there is no way to tell beforehand whether a sequence of operations will ever reach the last operation and stop. This is the halting problem.
He proved this by supposing the opposite: that there is an algorithm that can correctly determine if another given algorithm will ever finish. If you were to call this algorithm \textit{willHalt(algorithm)}, then you could construct the following algorithm:

\begin{verbatim}
HaltTest{
    A=1;
    B=2;
    C=3;
    ...
    If(willHalt(HaltTest)){
        While(1==1){
            Print("Still Going...");
        }
    } else
        Halt;
}
\end{verbatim}

Now if the HaltTest algorithm willHalt, then it goes on forever; and if it does not halt, then it stops. Since the willHalt(algorithm) algorithm cannot be correct, the premise is contradicted and the original premise – that there can be no such algorithm – is proven.

It is proven, that is, for the general case. There can be no such algorithm that will work in every case. However, there might be an algorithm for a special case. For instance, if we decided to restrict our algorithm above so that there could be no iterative structures (i.e. no ‘loops’ of any kind), no conditional or unconditional branches, goto statements
or recursion, but only straight line code, then for that case it can definitely be said that the algorithm will halt.

Since there is no algorithm that can solve the fully general case for the halting problem, it is called *undecidable*. Harrison in [6] demonstrated that if one were to solve the Safety problem for a given state of a general Access Control system, then one could also solve Turing’s Halting problem. The problem of safety in the general case is therefore undecidable, just like the halting problem in the Turing Machine. There can exist no algorithm that will correctly decide whether a given undesirable state can ever be reached in a protection system. In a more restricted case, however, the safety question can be rendered decidable. For example, if the commands in an HRU instance are restricted to mono-operational commands (i.e. commands that can only have one primitive operation), then safety for this special case of HRU is shown to be decidable.[6] It is also “essentially useless” [2], especially from an ORCON perspective, because a file’s creator cannot be established as its owner. Recall that the create command in our ORCON example has two operations.

In HRU, safety is shown to be undecidable for most variations/restrictions of the original model. [6] [1]
2.3 Sandhu

This result, described by Sandhu in [2] as “most disappointing,” has given rise to much research into the causes and solutions of the undecidability. He identifies that the expressive power of HRU to implement so many security policies is what limits its decidability. Sandhu introduces a protection system model that is based on HRU called the Typed Access Matrix Model [2]. In this model, each object is created with a certain type, such as (a file, user, security-officer, etc). Sandhu shows that this new restriction allows the model to implement more useful security policies whose safety is decidable.

Sandhu also proposes a Schematic Protection Model [9] in which the creation of subjects is shown to be a factor that limits decidability. The SPM model restricts subjects to certain types, and strictly limits the way that subjects can propagate these rights, especially transport rights. This model also uses graph representations of some elements of the model to simplify the algorithm by which decidable initial policies are identified.

2.4 Koch

Koch et al has in [10] proposed a graph-based protection system model that can only be altered by a set of transformation rules that conform to the given formal semantics for graph transformations. Each system must be configured with a set of rules designed
to express the desired security policies. Safety with respect to a given right leaking to an unauthorized entity is proven to be decidable in exponential time (worst case) by placing the restriction that the transformation rules that are configured can either add nodes and edges, or delete them, but not both. The authors prove this by establishing an upper bound on the number of transformations that need to occur in order for a certain right to leak from one vertex of the system to another. They find this upper bound by calculating the number of transformation rules, the number of subjects that could possibly invoke each of the rules, the number of objects that can be created, etc. and finding the length of the shortest sequence of transformation rules (the *derivation sequence*) that must occur to achieve the undesirable state. Then they check each sequence of transformation rules of that length for its presence. In order to bound the number of users, the system must have a finite list of subject labels (user names) configured. The same is true for objects.

This model solves the safety question for this kind of limited and self-contained system, but the design of the rules is left to the user of the system. The model itself does not provide any primitive operations or restrictions on the flow of rights. The authors give two examples: a Discretionary Access Control security policy, which was found to be unsafe, and a simple Role Based Access Control policy, which they show to be safe. Since the model comes with no operations or restrictions of its own, its usefulness is limited by the need to design the actual transformation rules by which it can be made to
provide useful security policies that can also be proven safe. It is also limited by the requirement that all of the users and objects be predetermined.

This model is of interest in that it renders the safety question algorithmically decidable - for a set of operations that has yet to be determined – by means of graph-based representation of the protection system’s state, using graph-transformation rules as operations, and by bounding on the number of subjects and objects in the system.

2.5 Lipton and Snyder

Lipton and Snyder in [5] formalize a protection system in which the access control data is represented as a directed, labeled graph with vertices representing subjects and objects, and labeled edges representing rights which the origin vertex has over the destination vertex. Refer to Figure 2-1. In this graph, $x$ can read $y$, and $y$ can write $z$. 
In their model, the operations allowed on this access control graph include *take* and *grant* to transfer rights from one vertex to another, and the model they describe has come to be known as the Take-Grant model. The operations are stated as Graph Rewriting Rules, sometimes known as Graph Transformations.

The Take rule is stated as follows:

Let $x$, $y$, and $z$ be three distinct vertices in a protection graph, and let there be an arc from $x$ to $y$ with label $\gamma$ such that $t \in \gamma$ and an arc from $y$ to $z$ with some label $\alpha \subseteq \{r, w, x, t, g\}$. The Take rule allows one to add the arc from $x$ to $z$ with label $\alpha$, yielding a new graph $G'$. This is represented graphically by Figure 2-2.
The transport rights in [5] are the read and write rights; however, most subsequent research restates the model with a Take right and a Grant right, which are the transport rights. [11] This avoids confusion and keeps the distinction between transport rights and generic rights. Any vertex that has a t right over another vertex (i.e. an edge labeled t) can Take any of the rights that it has. Similarly, any vertex that has a g right over another vertex can Grant to that vertex any rights that it has.

The remaining operations are defined as follows, from [5]

*Grant:* Let x, y, and z be distinct vertices in a protection graph G, and let there be an arc from x to y with label γ such that g ∈ γ and an arc from x to z with label α ⊆ {r,w,c}. The grant rule allows one to add an arc from y to z with label α, yielding a new graph G'. Intuitively x grants y the ability to do α to z.

![Figure 2-3](image-url)

*Create:* Let x be any vertex in a protection graph; create allows one to add a new vertex n and an arc from x to n with label {t,g,r,w,c}, yielding a new graph G'. Intuitively x creates a new user that it can take, grant, read, write, and call. In our representation

![Figure 2-4](image-url)
**Remove**: Let \( x \) and \( y \) be distinct vertices in a protection graph \( G \) with an arc from \( x \) to \( y \) with label \( \alpha \). The remove rule allows one to remove the arc from \( x \) to \( y \), yielding a new graph \( G' \). Intuitively \( x \) removes its rights to \( y \). The remove rule is defined mainly for completeness, since protection systems tend to have such a rule.

![Diagram of the remove rule](image)

Figure 2-5

The take-grant model provides restrictions on the movement of rights in the system by requiring a transport right (take or grant) be present in the subject to authorize such a move. Since the model restricts the flow of rights to connected vertices (subjects), the safety problem becomes figuring out which vertices are connected. For the take-grant and related models, that is very straightforward.

The graph representation lends itself to using graph algorithms and graph transformations in the operations of the system. This allows the application of graph theory to this problem space. The same kinds of restrictions and algorithms could have

---

1 There is also another operation that was defined in the original paper: **Call**. Call is not generally included in subsequent papers. Its semantics can be replaced by create and grant operations. It is defined as follows: Let \( x, y, \) and \( z \) be distract vertices in a protection graph \( G \), and let \( \alpha \subseteq \{ r, w, c \} \) be the label on an arc from \( x \) to \( y \) and \( \gamma \) the label on an arc from \( x \) to \( z \) such that \( c \subseteq \gamma \). The call rule allows one to add a new vertex \( n \), an arc from \( n \) to \( y \) with label \( \omega \), and an arc from \( n \) to \( z \) with label \( r \), yielding a new graph \( G' \). Intuitively \( x \) is *calling* a program \( z \) and passing parameters \( y \). The "process" is created to effect the call: \( n \) can read the program \( z \) and can \( \omega \) the parameters.
been proposed for a matrix representation; however, it may not have been apparent to do so when the system objects were not seen as connected vertices in a graph. This is one of the reasons that the use of graph representations for protection systems is appealing.

For the Take-Grant model, it has been shown that not only is safety decidable for this model, but the algorithms that decide whether the system is safe for a given right run in linear time. In other words, the running time of the algorithm is directly proportional to the size of the protection system, i.e. the number of system objects. That is quite an improvement over being undecidable!

Although the Take Grant model exhibits linear algorithms to decide if the system is safe for a given right, the decision is usually negative. It has the unfortunate property of allowing symmetric (reversible) transport of rights in all cases. Somewhat surprisingly, the flow is not restricted to the direction that the take or grant edge represents. [3] demonstrates that if two vertices are connected, the rights that each have are available to the other in a future state of the protection system. The logic is as follows: consider two vertices p and q; p has g on q (Figure 2-6 a).
Now p can create a new vertex p’, and in so doing, gets t and g on p’ as depicted in b. Vertex v can now grant g on p’ to q, resulting in c. Vertex q can now grant any of its rights to p via p’ – the reverse of the original direction of the flow. Thus, in the Take-Grant model, if a subject p is given ‘grant’ on another subject q, then q can obtain a ‘grant’ right by which to grant rights to p. In the final analysis, every node that is connected (i.e. has a take or grant) to another shares all of its rights with every other node in the connected portion of the graph. The only way to keep from sharing is be disconnected – that is, unable to take or grant any rights with any other object in the system. It does not pass the ‘useful’ test. These undesirable properties have limited the consideration of the principles illustrated by Lipton and Snyder.

2.6 Lockman and Minsky

In [3], Lockman and Minsky describe an alteration of Take-Grant, which they call Take-Receive, by which the undesirable reversibility inherent in the Take-Grant model is eliminated. Take-Receive has efficient algorithms by which to decide safety, but remains limited in expressive power. In their model, the flow of rights cannot take place
unless a right (either ‘take’ or ‘receive’) is held by the receiver of the transfer. The authors define several logical predicates based on the rights that the system objects possess. A predicate is a logical statement that contains one or more variables, sometimes called a propositional function. One of the predicates used in [3] is called $link(p,q)$. The definition of this predicate is as follows:

The predicate $link(p,q)$ is defined to be true in a protection state $S$ and for a pair of subjects $p,q$ in $S$ if and only if a MOVE of rights from $p$ to $q$ is authorized in $S$.

In the context of the Take-Grant model, $link(p,q)$ is true if $p$ has grant on $q$, or $q$ has take on $p$ (or both). In the Take-Receive model, however, the transport rights are Take, and Receive. In order for $link(p,q)$ to be true, either $q$ must have Take on $p$, or $q$ must have Receive on $p$ (or both).

It is important to note that in [3], although the authors depict the flow of rights using squares, circles and arrows, they avoid defining a representation of their protection system. The entire discussion of the protection system, including the predicates and subsequent proofs, are defined in terms of subjects, objects and their rights, without regard to the representation of those rights as a matrix, a graph, or some other data structure. In this way, they apply the predicate $link(p,q)$ (for example) to both the Take-Grant model and their own by defining when the predicate is true for that model. They also avoid formal semantics for operations, since they propose no representation of their model.
The Move operation can transport a right from some entity (either a subject or an object) \( p \) to some entity \( q \) if and only if \( q \) contains either Take or Receive rights over \( p \).

Move is designed to simplify the analysis, since either Send or Take will result in the same right going in the same direction between the same two entities.

The Create operation allows any subject \( p \) to create a new entity \( p' \), and gives \( p \) Take on \( p' \), and \( p' \) Receive on \( p \).

They define the \( path(p,q) \) predicate in terms of the \( link(p,q) \) predicate:

\[
Path(p,q) \text{ is true in a protection state } S \text{ if and only if there is a sequence of subjects } p = x_1, x_2, x_3, ..., x_n = q \text{ such that } link(x_i, x_{i+1}) \text{ is true for } i = 1, 2, ..., n-1.
\]

The authors address the safety problem by developing two more predicates: \( can-link(p,q) \) and \( can-path(p,q) \). These are defined to be true if the protection state \( S \) can be transformed into some state \( S' \) such that \( link(p,q) \) (or \( path(p,q) \) respectively) is true.

They demonstrate and prove that evaluating the can-path predicate is equivalent to deciding the subjects to which rights can flow – the safety question.

Intuitively, only those entities in the system that already have take or receive rights can ever be granted rights. This allows some separation between the different users of the system and the system is decidably safe. Once the initial state is determined, however, no rights can flow except to subjects who already have transport rights.
This is somewhat more reasonable than the Take-Grant system, but not very dynamic for those entities without transport rights.

2.7 Constraint Based Safety

Other research has focused on runtime constraints to enforce security policies. Jeager and Tidswell [4] argue that since it has been demonstrated (by HRU, for example in [6]) that the general safety question is undecidable, systems must either use a limited access control model or use constraints to verify that security policies are not violated at runtime. A constraint is a condition that will not be allowed to occur in the system. The runtime constraint check makes no guarantee about what may happen in a future state, only that for this particular operation, no defined system policy has been violated. An example might be that no member of the Red team can have read access to any object owned by a member of the Blue team. Instead of trying to design a limited access control model to prevent this from being possible, the constraint-based approach does not limit the operations. Instead, it allows any operations on the state of the system, but then checks state to verify that none of the constraints have been violated at each step along the way.

Their proposal is to simplify the constraint expression language and its implementation by using a graph-based representation of the constraints. This makes the expression of
constraints less complex and their run-time check more straightforward. By doing so, they propose a flexible access control model based on constraints that can be easily expressed by system administrators.

Their model, however, retains a high level of complexity. The authors state repeatedly that any extension of their model runs the risk of too much complexity. [4] It is left to the system administrator to choose the security policies that will be used in the system, and therefore such a system is only as ‘safe’ as the design of its policies. Such a model, in any case, can make no guarantees about future states; only that policies have not been violated. The safety question, as has been discussed up to this point, is not even asked by a constraint system such as this.

2.8 Next Steps

Sandhu has expressed his intuition [7] about the applicability of graphs and graph theory to this problem space, especially to the safety question. Citing Lockman and Minsky, Tidwell and Koch, he declares that reconciling the results of this more recent research in graph-based access control systems would be a step forward in understanding how graph theory and specific graph applications can provide solutions to the problems in this domain.
3 An Alternative

Some of the research outlined in previous sections has focused on finding the right set of restrictions for a model, to allow the safety analysis without limiting the power to perform useful operations too much. Other research has abandoned any inherent limitation on the model, relying instead on constraints to check and see if any policies are violated at runtime. I propose as an alternative a hybrid security model that extends the capabilities of the decidable models (such as take-grant) with runtime-enforceable constraints.

This hybrid system could be modeled as a graph and a set of operations that are decidable in terms of safety analysis. These operations would assure safety, but limit the ability to include some useful operations, such as granting a new right to an existing user. This capability will be provided by additional operations that are not parts of the underlying model. Enforcement for these additional operations will be achieved through the use of runtime constraints.

With this combination of a graph-based model and additional operations and constraints based on the underlying model, an access control system shall be modeled in this Thesis that will provide for both safety and usefulness.
For example, a hybrid model such as this would include an ‘addRight’ operation that only a trusted user is allowed to run. Unlike the constraint-based models mentioned in the previous section, this model need not simply enumerate the policies and check for a violation. Since the model is decidable, there is an algorithm to decide if there is any future state in which a right can leak to an unauthorized subject. This algorithm itself can be used as a constraint to provide a safety analysis of the ‘extra’ operation before its completion.

For example, suppose the following state in a take-grant model:

Figure 3-1

Recall that the safety analysis in the take-grant model can decide which nodes can ever get which rights. The nodes connected by a ‘t’ or ‘g’ edge (in either direction) can share all of their rights with each other. So the ‘possible’ graph becomes Figure 3-2.
What if the organization wanted vertex b to write to vertex z? This is not allowed under the restrictions of the take-grant system because vertex b does not have the take right for any vertex that has the w right on z. Since the algorithm that decides what the possible future states can be starts from a given state, this additional ‘extra’ operation can be allowed, and then analyzed for the ‘normal’ operations, to see if it is now possible to reach an undesirable state.

If w on z is given to b, Figure 3-3 is the result.
The safety analysis of this graph yields Figure 3-4.

In this case, the addition of the new right in b did not result in the possible future leakage any other right, because the w right is not a transport right. It is a “label that can be passed around, and that is all.” [5] This need not be guessed at, or reasoned about; it can be verified at runtime by the same algorithm that provides the safety analysis of the take-grant system in the first place. The graph algorithms can show the trusted user what the possible leaks are if he goes ahead with the ‘grant’. With the addition of runtime constraints that are based on the underlying model, useful operations on the access control system (which would not be allowable using the graph-based operations alone) can be part of a hybrid system. This allows the useful cases of the security policy to be safely included without losing the power and elegance of the graph representation as the basis of the security system.
In addition to specifying the hybrid model, it will be implemented for research purposes.

The research implementation of this model will provide an illustration of the model’s usefulness in applying these concepts to the safety domain.
4 The Model

The access control model being proposed by this Thesis shall be called the Graph Plus model. It is one of the contributions of this Thesis that the Graph Plus model is a realization, with formal semantics, of the Take-Receive model of Lockman and Minsky. [3]

In addition to implementing the transport rights and predicates of the Take-Receive model, the Graph Plus model incorporates elements of some of the other related work mentioned in previous sections. Graph Plus uses a graph representation of the state of the protection system, as does the Take – Grant, the Schematic Protection Model and others. It is important to note that unlike those, the edges do not represent rights held by one vertex over another, but instead represent the paths over which rights can be sent (or taken). Its safety algorithms are based on the predicates and proofs of [3]; however, the Graph Plus model adapts those predicates to its own graph-based representation of the state of the system.

The model also includes the ‘remove’ operation from Take-Grant and others, and the ‘own’ right to restrict the ‘remove’ operation, to enable the ORCON security policy. In Graph Plus, the rights in the system are modeled as Authorizations (called Tickets in the Schematic Protection Model [9]), which are privileges that can be exercised upon particular subjects or objects.
4.1 Representation

The Graph Plus access control model is represented as a directed, weighted graph in which the vertices represent system entities. The entities are divided into Subjects and Objects; the subjects are the active entities such as users and processes. The Objects are the passive entities that are acted upon by the subjects. In this model, the subjects and objects do not intersect.

In the Graph Plus model, the vertices themselves contain the rights that they have over one another. The edges represent links over which rights are allowed to be sent or taken among the vertices in the system. This is unlike other graph-based models such as the Take-Grant model, in which the edges contain the rights that one vertex holds over another.

Formally, the Graph Plus access control model is represented as a directed, weighted graph \( G = (V, E) \) where:

- \( V \) is a set of annotated vertices and \( E \) is a set of weighted edges.
- \( P = \{T, R, o\} \cup P_{\text{inert}} \) is the set of all privileges in the system. \( \{T,R\} \) are the transport privileges Take and Receive. The ‘own’ right restricts the removal of vertices (and their edges), and \( P_{\text{inert}} \) is the set of ‘inert’ privileges, e.g. read,
write and execute. These can be defined when the system is configured. For the model as instantiated in this Thesis, the set $P_{\text{inert}} = \{ r, w, x \}$.

- $T_v = \{ u, a, o \}$ is the set of Vertex types, i.e. Administrator, User, Object. $T_{\text{sub}} = \{ a, u \}$ is the set of Subject Types. In the visualizations that follow, User subjects are squares and objects are circles. The administrator vertex is a hexagon. A vertex can be either a Subject (an Administrator is a special kind of Subject) or an Object, but not both – unlike HRU and many other Protection System models.

- Let $p$ denote a privilege, and $v$ a vertex. The term $p:v$ is called an authorization. Informally, $p:v$ can be read as “privilege $p$ on vertex $v$.” For example, $r$:MyFile would represent an Authorization to read the file ‘MyFile’.
  
  - Let $a = p:v$ denote an authorization. The function $\text{priv}(a) = p$, and the function $\text{vertex}(a) = v$

- Each vertex $v = (\text{name}, \text{type}, A)$ where name is the name of the vertex, type is the type of the vertex, and $A$ is the (possibly empty) set of Authorizations possessed by the vertex. For example, suppose vertex $v1$ has read privileges over $v2$ and $v3$. In this case, $r:v2$ and $r:v3$ would belong to the set of authorizations of $v1$.
  
  - For a vertex $v = (n, t, A)$, the function $\text{type}(v) = t \in T_v$

  - For a vertex $v = (n, t, A)$, the function $\text{auth}(v) = A$, and $\forall a \in A, \text{priv}(a) \in P$ and $\text{vertex}(a) \in V$. 
- \( Q = \{ \text{Strong, Weak} \} \) denotes the weight of an edge between Vertices. Strong edges are depicted as blue solid edges, and Weak edges as red dashed edges.
- Each edge is a directed, weighted edge with no label, modeled as \( ((v1, v2), q) \) where \( (v1, v2) \) is an ordered set of vertices and \( q \in Q \).
  - Let \( e = ((v1, v2), q) \) denote a weighted edge. The function \( \text{quality}(e) = q \).

Figure 4-1 depicts a Graph Plus protection state graph.

In this system, the vertices represent system entities such as users, processes, and files. Subject vertices represent those entities that can exercise volition to Take or Send authorizations to other entities, such as users and processes. Object vertices represent those entities that are passive with respect to initiating the transfer of authorizations, but which can be given authorizations, such as files and system utilities.
The edges of the graph represent the possibility of the flow of authorizations between vertices. An edge will be inserted in the graph whenever a vertex is given a Take or Receive Authorization over another vertex. The presence of an edge denotes a link over which authorizations can move within the access control system. The edges do not represent rights that vertices have over one another, as in the Take-Grant model. The edges encode the presence of a Transport Right in the receiving vertex, so that more efficient algorithms may be used to calculate the possible future states of the system.

Keep in mind that safety analysis follows the principle of conservative or worst-case analysis: it does not matter what the users and processes in a system intend to do, or what you think they will do. It matters only what it is possible for them to do.

### 4.2 Operations

The operations, which can only be invoked by the subject vertices in the system, are defined as follows.

Create(caller, name, type) creates a new vertex in the protection state:

Let:

- \( S = (V,E) \) denote the current state of the protection system
- \( \text{caller be a vertex} = (\text{name}_c, \text{type}_c, A_c) \)
- \( \text{Let } v' \text{ be a vertex} = (\text{name}, \text{type}, \{R:\text{caller}'\}) \)
Let caller' be a vertex = \((\text{name}_c, \text{type}_c, A_c \cup \{T:v', o:v'\})\)

If caller ∈ V and type\(_c\) ∈ T_{sub} and type ∈ \{s, o\} then:

Create(caller, name, type) produces a new Protection State \(S' = (V', E')\) where

\[V' = (V - \text{caller}) \cup \{\text{caller}', v'\}\]

and

\[E' = E \cup \{(\text{caller}', v'\}, \text{Strong}), ((v', \text{caller}'), \text{Strong})\}\]

Otherwise, Create(caller, name, type) fails and the protection state \(S\) is unchanged.

Our initial state is depicted in Figure 4-1. After calling Create(node4,S), the new state is Figure 4-2.
The Take(caller, p, on, from) intuitively \textit{takes} the Authorization \textit{p:on from} the vertex \textit{from}. For example, Take(vertex0, w, vertex1, vertex2) would take the Authorization \textit{w:vertex1} from vertex2 and give it to the caller. The Authorization is left intact in vertex2.

Formally:

Let:

- $S = (V,E)$ denote the current state of the protection system
- caller be a vertex $= (\text{name}_c, \text{type}_c, A_c)$
- Let \textit{on} and \textit{from} be the vertices $(\text{name}_{on}, \text{type}_{on}, A_{on})$ and $(\text{name}_{from}, \text{type}_{from}, A_{from})$, respectively

If $\{\text{caller, on, from}\} \subseteq V, \text{type}_c \in T_{sub}, p \in P, p \not\in \{o\}, p:on \in A_{from}$ and $T:from \in A_c$ then

Take(caller, p, on, from) produces a new Protection State $S' = (V', E')$ where

$V' = (V - \text{caller}) U \{ (\text{name}_c, \text{type}_c, A_c U \{p:on\}) \}$ and

If $p = T$ or ($p = R$ and $\text{type}_{on} = s$) then $E' = E U \{ ((\text{name}_{on}, \text{name}_c), \text{Strong}) \}$

If $p = R$ and $\text{type}_{on} = o$ then $E' = E$

Otherwise, Take(caller, p, on, from) fails and the protection state $S$ is unchanged.

Send(caller, p, on, to) copies an authorization from the caller vertex to the to vertex:

Let

- $S = (V,E)$ denote the current state of the protection system
• caller be a vertex = (name_c, type_c, A_c)

• p be a character

• on and to be the vertices (name_on, type_on, A_on) and (name_to, type_to, A_to), respectively

If caller ∈ V, on ∈ V, to ∈ V, type_c ∈ T_{sub}, p ∈ P, p ≠ o, p:on ∈ A_c and R:caller ∈ A_to then:

Send(caller, p, on, to) produces a new Protection State S' = (V', E') where

V' = (V – to) U { (name_to, type_to, A_to U {p:on}) } and

If and (p = T or p = R) and type_on ∈ T_{sub} then E' = E U { ((name_on, name_to), Strong) }

If p = T and type_on = o then E' = E U { ((name_on, name_to), Weak) }

If p = R and type_on = o then E' = E

Otherwise, Send(caller, p, on, to) fails and the protection state S is unchanged.

If node1 were to Send(node1, r, node2, node4), Figure 4-3 would result.
Sending a r:node2 authorization to node4 grants that vertex the ability to read the node2 object, a file for example. It does not result in an edge being added to the graph, because no new path for the flow of authorizations has been added. Information can flow between node2 and node4, but there is no channel for authorizations to flow.

Using this model, the only way that authorizations can flow between nodes is if a Take or Receive authorization resides in the receiver.

In our example, if node1 were to Send(T, node4, node2), then since node2 already has a Receive:node1, the Send would succeed and node2 would get Take on node4. This does create a new channel for the flow of authorizations, as depicted in Figure 4-4.
Remove(caller, v) removes a vertex from the protection state:

Let

- $S = (V, E)$ denote the current state of the protection system
- caller be a vertex = $(\text{name}_c, \text{type}_c, A_c)$
- $v$ be a vertex = $(\text{name}_v, \text{type}_v, A_v)$
- $E_v$ be the set of all edges $((v_1, v_2), q)$ in $E$ for which $v_1 = v$ or $v_2 = v$
- $A_r$ be the set of authorizations $p:v$ for all $p \in P$

If caller $\in V$ and $v \in V$ and type$_c \in T_{\text{sub}}$ and the authorization $o: \text{name}_c \in A_c$ then:

Remove(caller, v) produces a new Protection State $S' = (V', E')$ where:

- $V' = \text{the set of vertices } v' = (\text{name}', \text{type}', A')$ such that $\forall v_i \in (\text{name}_i, \text{type}_i, A_i)$ in $V - v$, $\text{name}' = \text{name}_i$, $\text{type}' = \text{type}_i$, and $A' = A_i - A_r$
- $E' = E = E_v$

Otherwise, Remove(caller, v) fails and the protection state $S$ is unchanged.
Remove is included for completeness, but since removing vertices and their edges cannot create new paths over which authorizations can be Sent or Taken, Remove is not involved in any analysis of safety.

4.3 Weak Edges

The protection state now contains a ‘weak’ edge between node4 and node2. Since node2 is an object node, and does not have the volition to invoke the Take operation, the edge is considered weak. It is possible, however, for another node to obtain the Take authorization over node2, and thus get a Take authorization over node4. For this reason, the weak edges must be included.

Consider the following scenario of Figure 4-5. Vertex 2 is an object with a Take authorization on vertex 3. Since vertex 2 is an object, it cannot exercise the volition to invoke the Take(…) operation on vertex 3. But if vertex 0 Takes T:vertex2 from vertex1, then you have the graph of Figure 4-6. Now vertex0 has access to the T:vertex3 that vertex2 contains – even though vertex2 is an object, and cannot exercise any volition. If vertex0 gets T:vertex3 from vertex2, the result is the graph in Figure 4-7.
Figure 4-5
Figure 4-6
Now vertex0 can take the w:vertex4 authorization from vertex3, even though an object intervened on the path. Without the weak edge, the path from vertex3 to vertex0 would not have been included in the possible paths over which authorizations could flow.
On the other hand, the path from vertex3 to vertex2 cannot be a strong edge, because if there is no subject capable of invoking Take, then there truly is no path over which authorizations can flow through that object.

It is not necessary, however, to include an edge between a subject vertex and an object vertex when the subject has a Receive authorization over the object; the object will never have the ability to invoke Send, no matter what vertex obtains the Receive authorization.

### 4.4 Safety

In the Graph Plus model, an edge is created in the protection state only when a Send or Take operation that involves a T or R privilege is performed, or when a Create operation (which creates the T:v' and R:caller authorizations) is performed. In order for a Send or Take operation to succeed, the Receive or Take authorization, respectively, on the source vertex (and its corresponding edge) must be present in the receiving vertex. In the Graph Plus model, this also means that an edge already exists between the source and receiving vertices.

In order for a vertex to have an edge to a particular vertex in any future state, it must obtain a Take or Receive authorization (and the corresponding edge) to that vertex. For that to happen, there must already exist a path between it and the source vertex; that
is, a sequence of vertices linked by edges. Furthermore, using Create cannot complete a path that did not previously exist since Create only creates edges between the creator and created vertices.

Therefore, the model and its operations cannot introduce a new path for authorizations to flow that was not there when the system was defined. Since safety is defined as being unable to reach a state in which an authorization “leaks” to a subject that should not have it, the system is decidably safe.

The formal proofs of this are given in [3] and are fully applicable to this model because they are proven in terms of predicates that are not strictly related to the underlying model, but only to authorizations belonging to subjects. They are easily modified to refer to the Graph Plus model’s design. I have outlined them here informally, and have included the formal restatement of them in terms of this model in Appendix A – Proofs of Safety Claims for the interested reader.

### 4.5 Usefulness

So far, our access control system is very safe, but not very useful. Using some of the figures above, and walking through the ‘Create’, ‘Send’ and ‘Take’ operations shows that each node can create children, and the children can create children, and the authorizations that each sub-graph has are completely isolated to that sub-graph. Yet,
each parent can obtain and communicate any authorizations to or from any of his
children or grandchildren. From a safety standpoint, each separate sub-graph in the
initial state of the system can generate a fully connected subgraph, and that is all.

This is not as bad as it may appear at first. Many organizations are composed of
separate departments, for which it is desirable that the department head has control
over all of the subordinates. This model does implement the ORCON security policy.
However, we would like to include some other useful policies. Our example will be to
add an authorization to a user arbitrarily.

Let us add to our system a trusted user, called ‘admin’. Such a user can be given access
to an additional set of operations in our hybrid system, hereafter called admin
operations. The previously discussed operations shall be called the user operations.
The admin operations are enabled only in an admin vertex. The admin operations are
defined as follows.

addAuthorization(caller, p, on, to): creates an authorization in the to vertex:

Let:

- $S = (V, E)$ denote the current state of the protection system
- caller be a vertex $=(\text{name}_c, \text{type}_c, A_c)$
- $p$ be a character
- $on$ and $to$ be the vertices $(\text{name}_{on}, \text{type}_{on}, A_{on})$ and $(\text{name}_{to}, \text{type}_{to}, A_{to})$
If caller ∈ V, on ∈ V, to ∈ V, p ∈ P, and type_c = a then:

```
addAuthorization(caller, p, on, to) produces a new Protection State S' = (V', E')
```

where

```
V' = (V – to) ∪ \{ (name_to, type_to, A_to U \{p:on\}) \} and
if and (p = T or p = R) and type_on \in T_{sub} then E' = E \cup \{ ((name_on, name_to), Strong) \}
if p = T and type_on = o then E' = E \cup \{ ((name_on, name_to), Weak) \}
if p = R and type_on = o then E' = E
```

Otherwise, Take(caller, p, on, from) fails and the protection state S is unchanged.

Consider the protection state in Figure 4-8

![Protection State Diagram](image-url)
The admin user gains access to the Graph Plus system as one of the subjects in the system. Unlike the user operations however, the addAuthorization operation can add a authorization to any node in the system, not just the admin node. If node1 is an administrator vertex, and runs addAuthorization(node1, Take, node4, node3) then the protection state of Figure 4-9 is the result.

![Protection State Diagram]

The removeAuthorization(caller, p, on, from) operation removes the given authorization and any edges that are associated with it:
Let:

- $S = (V,E)$ denote the current state of the protection system
- caller be a vertex $= (name_c, type_c, A_c)$
- $on$ and $from$ be the vertices $(name_{on}, type_{on}, A_{on})$ and $(name_{from}, type_{from}, A_{from})$
- $from'$ be the vertex $(name_{from}, type_{from}, A_{from} \setminus \{p:on\})$
- $E_{onFrom} = \{(v_1, v_2, q) \in E \mid v_1 = on \text{ and } v_2 = from\}$

If $\{\text{caller, on, from}\} \subseteq V$ and type$_c = a$ then:

RemoveAuthorization(caller, $p$, on, from) produces a new Protection State $S' = (V', E')$ where

$V' = V \setminus \{\text{from}\} \cup \text{from}'$ and

If $R:on \in \text{auth}(\text{from}')$ and type$_{on} = s$ then

$E' = E - E_{onFrom} \cup \{(name_{on}, name_{from}), \text{Strong}\}$

If $T:on \in \text{auth}(\text{from}')$ then

If $\text{type}_{from} = o$ then $E' = E - E_{onFrom} \cup \{(name_{on}, name_{from}), \text{Weak}\}$

Otherwise $E' = E - E_{onFrom} \cup \{(name_{on}, name_{from}), \text{Strong}\}$

Otherwise $E' = E - E_{onFrom}$

Otherwise, RemoveAuthorization(caller, $p$, on, from) fails and the protection state $S$ is unchanged.
It is possible for a vertex to have both Take and Receive authorizations over the same
node (e.g. T:v, R:v), but in this case Graph Plus only models one edge between the
nodes. Because an edge represents the possibility of authorizations moving between
the two vertices, only one edge is necessary. If either a Take or a Receive authorization
remains in the from vertex after the p:on is removed, the appropriate edge must be
calculated, as it is in the Send and Take operations.

The removeAuthorization(caller, priv, on, from) operation is included for completeness,
but has no role in the safety analysis since it does not create new paths over which
authorizations can flow.

The remaining two components of the Graph Plus model are the analysis functions
Path(from, to) and Closure(). These functions act as a constraint on the admin user’s
capability to create rights in the system arbitrarily.

Path(from, to) determines if there is a path along which rights can flow in the current
protection state:

Let \( S = (V,E) \) denote the current state of the protection system.

If \( t o \in V, \) \( f r o m \in V \) and

there is a sequence of vertices \( f r o m=x_1, x_2,...,x_n=to \) in \( V \) and

for \( i = 1,2,...,n-1 \), there exists an edge \( ((v_1, v_2), q) \in E \) | \( v_1 = x_i \) and \( v_2 = x_{i+1} \) and
quality ((x_{n-1}, to), q) = \text{strong}

then the Path(from, to) operation returns true

Otherwise, it returns false.

The path(from, to) operation returns true if there is a *path* from the *from* vertex to the *to* vertex. This makes use of the transitive closure algorithm that is available because of the graph representation of the protection state, and the definitions of the operations that have been included in the model. It can calculate whether there is any path (including weak edges) by which a authorization can flow from the *from* vertex to the *to* vertex. By the proofs outlined earlier, that also means that it can calculate if a path between two vertices can be created by a sequence of operations consisting entirely of user operations.

Recall that the safety question for protection systems asks, “Can a given undesirable state can ever be reached using the operations of the system?” This model renders the safety question decidable because of the way in which the propagation of authorizations is restricted. The Graph Plus model satisfies the following property: in order for authorizations to flow between vertices *p* and *q*, there must already exist a path between them. This is an invariant of this protection system.
The Graph Plus model calculates the transitive closure of the graph to identify paths in the protection state. Transitive closure in general means that if vertex $a$ has an edge to vertex $b$, and vertex $b$ has an edge to vertex $c$, then an edge from vertex $a$ to vertex $c$ can be created. For Graph Plus, because there are weak and strong edges, transitive closure means that if $a$ has an edge of any kind to $b$, and $b$ has a Strong edge to $c$, then a Strong edge can be created from $a$ to $c$. Recall that a Strong edge simply means that a Subject vertex is in the proper position to perform the Send or Take operations necessary for authorizations to move within the protection state. If $a$ has an edge to $b$, and $b$ has a Weak edge to $c$, then a Weak edge can be created from $a$ to $c$. A Weak edge represents the possibility of authorizations being transported if there is a Subject further down the path in the proper position. The transitive closure graph contains an edge from each vertex directly to every other (currently existing) vertex for which it will ever be possible to create one, using the user operations.

This property allows the safety question to be answered. The closure graph can be used to determine if any given node can ever receive a given authorization in some future state.

Closure() calculates the transitive closure of the edges in the graph, taking the weak edges and subject position into account:

Let $S_1 = (V, E)$ be the current protection state
For each protection State $S_i$ where $i$ is in \{1, 2, ..., n\}

Let $S' = (V', E') = S_i$

\[ \forall (v_1, v_2, v_3) \in V' \mid ((v_1, v_2), q_1) \in E' \text{ and } ((v_2, v_3), q_2) \in E' \]

If $q_2 = \text{strong}$ then $E' = E' \cup \{((v_1, v_3), \text{strong})\}$ otherwise

$E' = E' \cup \{((v_1, v_3), \text{weak})\}$

If $S' = S_i$ then the Closure() function returns $S'$ as the Closure Graph

Otherwise, let $S_{i+1} = S'$ and continue

The Closure() function creates a copy of the protection state, and calculates the transitive closure of the entire graph, with weak edges and subject position taken into account. It gives a graphical representation of the potential connections in the protection state.

Consider a protection state as depicted in Figure 4-10.
If the admin user runs Closure(), he can see every path as a direct connection, as in Figure 4-11.
This is a fairly connected graph, but notice that no authorizations can move from node0 to any other node.
Now suppose that our organization wished to allow node3 to get some of the authorizations possessed by node0. If the admin user runs `addAuthorization(node1, R, node0, node3)`, the graph in Figure 4-12 is the result.

![Protection State](image)

**Figure 4-12**

This does not look like a big change, but if the `Closure()` method is run again, we get Figure 4-13.
The result is a fully connected graph! Every vertex has a direct link to every other vertex. This may not be what the organization intended. This illustrates the use of the closure method to act as a constraint at run time to decide whether the anticipated change can ever lead to an undesirable protection state.
The admin user can also use the \texttt{path(node3, node1)} operation as a constraint to
determine if the protection state now has a path between to specific vertices. For
example, instead of running ‘closure’ and examining the entire closure graph, he can run
\texttt{path(node0, node3)} to ascertain if the anticipated operation makes it possible for
authorizations to ever flow from node0 to node3. In this case, it would return true.

In the underlying model, the closure algorithm is used to demonstrate that there can be
no safety violation in a future state, given the initial state and the user operations
included in the model. In the extended system, the closure algorithm is used at runtime
to answer the question, “is there a violation now?” In our scenario, the new state can
be abandoned or saved, depending on the safety analysis of the state with its newest
addition. With this constraint, the addAuthorization operation can be added to a hybrid
system to extend its expressive power and provide a useful tool for determining safety
in access control.

The preceding example shows how the \texttt{Closure()} or \texttt{Path()} functions can be used to
indicate that an addAuthorization operation might be ill advised. For an example in
which the addAuthorization operation could be used safely, consider Figure 4-14.
A likely scenario is that the other subject nodes in the system need to be able to read and write the file (node2) created by another user. Using the user operations of the Graph Plus model, there is no way to give node3 access to node2; however, using the hybrid system, the administrator can use addAuthorization(node1, r, node2, node3) to give node3 permission to read node2 (Figure 4-15).
There is no edge added because the read authorization is not a Transport authorization; no new paths have been added. The Closure() function can be run to show (Figure 4-16)
that there is no safety concern with this addition, and the administrator can save this change.

This example is an illustration of a Mandatory Access Control (MAC) Security Policy. In a MAC policy, a central authority determines the rules by which access to the resources in a system are granted to users of the system. This policy cannot be expressed using the user operations alone, nor by the Take-Receive model. It can be expressed by Take-Grant and HRU, but not safely. The Graph Plus model includes this security policy with a guarantee that no unsafe state can be reached by the user operations, and allows the administrator to make an informed decision. This is a contribution to the access control domain.

5 A Useful Illustration

The model described has been implemented in Java as a research and demonstration application called graphPlus. It uses a freeware graph implementation package called JGraphT, and a visualization package called mxgraph. It includes the model and its graph-based representation, the basic operations, the trusted user extensions, and the capability to save its state for future runs. With it can be simulated the autonomous actions of the various users in a system, including the admin user.
5.1 The Model's Representation

The model's representation of the protection state is a directed weighted multi-graph with no self-loops. This type of a graph is available from the open-source JGraphT library (http://jgrapht.org/). The only modification necessary to the graph object from the JGraphT package was to extend it with the Listenable interface. This gives it the capability to fire events when edges and vertices are added and removed, to support the visualizations. The primitive methods on the graph data structure are addVertex, removeVertex, addEdge, removeEdge. The following sections describe the aspects of the implementation that are the most important and/or interesting.

5.2 Adding Rights

Recall that in the Graph Plus model, the rights that a user has over other objects in the system are modeled as Authorizations in the form of an ordered pair (privilege : vertex). The meaning is that the vertex (user) that has this Authorization can apply the privilege to the identified vertex. For example, if vertex v1 has the r:v2 Authorization, then v1 can ‘read’ v2.

This is modeled in this way because the rights must reside in the vertices themselves. The edges do not have labels with rights as in Take-Grant; the edges only represent the possibility of the flow of rights in the system, so the rights must be associated with the entities over which they may be applied. Therefore, the implementation must do the
bookkeeping so that the edges are created or destroyed when the appropriate rights are
sent, taken, added, or removed. Figure 5-1 is the addAuthorization implementation.

```java
public PersistentEdge addRight(Vertex v, Right r)
{
    PersistentEdge edge = null;

    // if v is an object, adding take to it creates a pseudo-link ala Lockman & Minsky
    // for graphPlus, it is modeled as a weak edge

    v.addRight(r);

    if (!r.getPrivilege().equals(Right.TAKE)) {
        // if the right is 'take', then the direction of the flow of
        // rights is from the target to the source (the node that owns the right)
        edge = model.addEdge(findVertex(r.getVertexId()), v);

        if (v.getType().equals(OBJECT)) {
            model.setEdgeWeight(edge, 0);

            // There is no listener for the edge weight; in order to get the visualization
            // to respond to the new edge weight, we must remove the edge and add it back again
            // so that the addEdge event will fire
            model.removeEdge(edge);
            model.addEdge(findVertex(r.getVertexId()), v, edge);
        }
        else if (r.getPrivilege().equals(Right.RECEIVE) &&
                findVertex(r.getVertexId()).getType().equals(SUBJECT)) {
            // if the right is 'receive', then the direction of the flow of
            // rights is from the target to the source
            edge = model.addEdge(findVertex(r.getVertexId()), v);
        }
    }

    return edge;
}
```

Figure 5-1

5.3 Closure

Keeping track of the strong and weak edges allows the safety algorithm to be little more
than the transitive closure of the graph. All that is needed to add to the routine that
comes with the JGraphT package is the detection of weak edges (representing the
presence of an object in the path), and the position of a subject to take advantage of
any authorizations possessed by an object. The closure algorithm is listed in Figure 5-2. It runs in $O(n \log n)$ time.

```java
public <U extends Vertex, E extends PersistentEdge>
void closeGraph(ListenableDirectedWeightedMultigraph<V, E> graph)
{
    Set<V> vertexSet = graph.vertexSet();
    Set<V> newEdgeTargets = new HashSet<V>();
    Set<V> newPseudoEdgeTargets = new HashSet<V>();

    // At every iteration of the outer loop, we add a path of length 1
    // between nodes that originally had a path of length 2. In the worst
    // case, we need to make floor(log |V|) + 1 iterations. We stop earlier
    // if there is no change to the output graph.

    int bound = computeBinaryLog(vertexSet.size());
    boolean done = false;
    for (int i = 0; !done && i < bound; ++i) {
        done = true;
        for (V v1 : vertexSet) {
            newEdgeTargets.clear();
            newPseudoEdgeTargets.clear();

            for (E v1OutEdge : graph.outgoingEdgesOf(v1)) {
                V v2 = graph.getEdgeTarget(v1OutEdge);
                for (E v2OutEdge : graph.outgoingEdgesOf(v2)) {
                    V v3 = graph.getEdgeTarget(v2OutEdge);

                    if (v1.equals(v3)) {
                        // Its a simple graph, so no self loops.
                        continue;
                    }

                    if (graph.getEdge(v1, v3) != null) {
                        // There is already an edge from v1 --- v3, skip.
                        continue;
                    }
                    if (graph.getEdgeWeight(v2OutEdge) > 0) {
                        // Tests for a strong edge
                        newEdgeTargets.add(v3);
                    } else if (graph.getEdgeWeight(v2OutEdge) == 0.0) {
                        // Tests for a weak edge
                        newPseudoEdgeTargets.add(v3);
                    }
                    done = false;
                }
            }

            for (V v3 : newEdgeTargets) {
                graph.addEdge(v1, v3);
            }
        }
    }
}
```
Figure 5-2

```java
for (V v3 : newPseudoEdgeTargets) {
    E pEdge = graph.addEdge(v1, v3);
    graph.setEdgeWeight(pEdge, 0);
}
```
5.4 Visualization

The JGraphT package comes bundled with a visualization package called mxGraph. The mxGraph package renders the JGraphT graph object in a java applet. It included hooks for the listeners for a JGraphT graph, so that changes to the graph can be visualized in an event-driven fashion. The graphPlus research implementation makes use of the mxGraph package to visualize the protection state of the system.

5.5 Persistence

The state of the protection system modeled by the graphPlus research implementation can be saved to the file system at any time. This allows the graphPlus to simulate the autonomous action of many users upon the protection state, and allows research to be conducted in more than one session. The persistence is accomplished by means of JSON (Java Script Object Notation), which is a compact notation to represent the state of complex objects.
6 Conclusions

This thesis has consolidated disparate research findings on graph-based applications to safety in access control systems. With the results of this research, a graph-based representation of an access control system and a set of operations and constraints have been modeled. The model’s user operations are restricted to those that render the model decidable in terms of safety analysis. To provide more expressive power and usefulness, additional operations are allowed which are not parts of the underlying model, but the model and its algorithms are used to provide a safety analysis of the additional operation before its completion. With the addition of this constraint, which is decidable, useful operations on the access control system (which would not be allowable using the graph-based operations alone) are now part of a hybrid system.

With this combination of a graph-based model and additional operations and constraints based on the underlying model, a research implementation has been developed which can be used as an illustration for a general-purpose access control system or as a user application to enhance the underlying operating system’s access control features.

The Graph Plus model shows that the method of using a hybrid system of restrictions and constraints based on graph theory allows the inclusion of an operation such as
addAuthorization and the ability to tell whether that operation will ever lead to an unwanted state. This is a useful and needed application of the principles of graph theory to the access control domain.

The insight gained by the implementation of the model, and the resulting simulation activities, includes a crucial aspect of the graph representation itself. While the notion of strong and weak links between vertices is mentioned in [3] (they call them pseudo-links), the characterization of them as weighted edges, and the subsequent implementation of the transitive closure algorithm that takes the weight into account, is a by-product of the research implementation of the model.

The simulation activities have been particularly useful in characterizing the expressive power of the underlying graph-based model and its basic operations. Without such a simulator, it would be more difficult for the theoretician to realize the ramifications of the model’s restrictions.
6.1 Future Work

Using a trusted user as a means of extending the expressive power of the hybrid approach is only one way to illustrate its usefulness. Since the model constrains the addAuthorization operation by means of the closure algorithm, it would be possible to develop a system of security policy invariants, and to check those at run time. For example, policy statements such as ‘a red team member cannot obtain a transport authorization over a blue team member’ can be characterized and expressed in a way that can be included in the model. In this way, the system can enforce the policy rules developed by the organization even when using the hybrid, or constraint-based operations. The system users could be given that operation in that case, and no admin user would be needed. This version would be an avenue for future investigation.
6.2 References


[8] A. M. Turing, "On computable numbers, with an application to the


Appendix
7 Appendix A – Proofs of Safety Claims

This proof is modified in its terminology and context from that which is given in [3]; otherwise, it is stated substantially in their own words. Their numbering has been retained for easy comparison. The terminology and definitions specific to the Graph Plus model refer to section 4.1 of this Thesis.

Predicates:

- \( \text{link}(p,q) \) is true in a protection state \( S \) if and only if there exists a strong edge between \( p \) and \( q \).

- \( \text{path}(p,q) \) is true in a protection state \( S \) if and only if there is a sequence of subjects \( p = x_1, x_2, \ldots, x_n = q \) such that \( \text{link}(x_i, x_{i+1}) \) is true in \( S \) for \( i = 1, 2, \ldots, n-1 \).

- \( \text{can-link}(p,q) \) is defined to be true in a protection state \( S \) if and only if \( S \) can be transformed into some state \( S' \) such that \( \text{link}(p,q) \) is true in \( S' \).

- \( \text{can-path}(p,q) \) is true in protection state \( S \) if and only if \( S \) can be transformed into some state \( S' \) such that \( \text{path}(p,q) \) is true in \( S' \).

- Note that the predicates \( \text{path} \) and \( \text{can-path} \) are both reflexive and transitive.

Due to the assumed passivity of objects, a Take or Send of rights from an entity \( p \) to an entity \( q \) may occur only if at least one of the following conditions is satisfied:

1) \( q \) is a subject and possesses \( T:p \); (i.e. there is a strong edge from \( p \) to \( q \)) or

2) \( p \) is a subject and \( q \) possesses \( R:p \) (i.e. there is a strong edge from \( p \) to \( q \)).
These, then, are the only cases where link(p,q) is true, but they are not the only cases of interest. In both situations above there is a subject in the proper position (receiver for the take right, sender for the receive right) to exercise the volition required by the right for a right to flow from p to q. If q is an object, however, and possesses T:p (whether p is a subject or an object) then q, being an object, cannot exercise the volition needed to use the T:p right which it possesses; thus link(p,q) is false here. Since, however, in some future protection state, this right may be transferred to some subject that can use it, we must include the potential effects of such currently unusable rights in our analysis.

We introduce two new predicates to include such a situation:

- **pseudo-link(p, q)** is true in a protection state S if and only if either:
  - link(p,q) is true; or
  - there is a weak edge from p to q (i.e. q is an object holding T:p)

- **pseudo-path** is the transitive closure of pseudo-link (i.e. a sequence of pseudo-links)

Thus pseudo-link(p,q) is true for all of the cases examined so far.

Finally, there is the case of some entity q (which can be a subject or an object) that holds a receive right over some object p. Since the use of a receive right to authorize a
Send requires volition on the part of the sender, a receive right over an object can never be used; no matter who possesses the right, the object can never exercise the necessary volition to use it. We shall therefore assume that receive rights over objects do not exist, since they have no effect on present or future channels for the flow of rights.

Theorem 4.1: For entities p and q,

\[ \text{can-path (p,q) } \iff \text{there exists some subject } u, \]

(possibly identical to p or q)

\[ \text{such that pseudo-path (p,u) and link (u,q).} \]

The result is that a necessary condition to create a path from p to q is the prior existence of a chain of strong and weak edges, i.e., a pseudo-path, from p to q. The additional requirement that a subject be the last or next to last entity in the chain provides the minimal element of volition necessary to turn this pseudo-path into a path.

Proof: For brevity in this proof we shall use the predicate \( \text{cond(p, q)} \) as shorthand for the condition "there exists some subject u such that pseudo-path(p,u) and link(u,q)" (i.e., the right-hand side of this theorem). First we shall prove that \( \text{can-path(p,q)} \) implies \( \text{cond(p,q)}. \)
Now if can-path(p,q) is true in some protection state S, then, by definition, path(p,q) is true in some state S' derivable from S. It is easy to see that cond(p,q) holds in this state S'. First, if q is a subject, then we let u be q, and, since every path is a pseudo-path and link is reflexive, cond(p,q) holds in S'. If q is an object, then the entity before it on the path must be a subject, since we can never have a link between two objects. We call this next to last entity u, and again, since path(p,u) (the rest of the path in S') then pseudo-path(p,u), and cond(p,q) again holds in S'. To complete this half of the theorem we shall use this fact to prove that cond(p,q) must also hold in the initial state S. We shall do so in a fashion similar to the proof of Theorem 3.1, by demonstrating that no operation can make cond hold if it did not do so in the preceding protection state, i.e., that cond is invariant under our transport operators. Any transport operation is either a Send, Take or a CREATE operation. Since CREATE only establishes communications between an existing and a new subject, clearly no CREATE operation can establish cond(p,q) for previously existing p and q. Thus we only need consider whether a Send or Take operation can do so. Consider a protection state A such that: a) cond(p,q) is false in A; and b) cond(p,q) is true in state A' which is the result of a single Send or Take operation invoked in A. There are two possible ways in which this can happen: either 1) this Send or Take adds link(u,q) for some subject u; or 2) it adds a pseudo-link between some other pair of entities, say from entity $x_i$ to $x_{i+1}$. Figure 7-1 depicts state A for case 1.
For this case the following must be true in state A: pseudo-path (p,u); some entity r must hold either R :u or T:u; and link (r,q). Since r holds a right over u in state A, pseudo-link(u,r) is true in A, and therefore, by transitivity, so is pseudo-path(p,r). Since link(r,q), at least one of r and q must be a subject. We let the one that is play the role of subject u in the cond predicate (recall that u is an existentially quantified bound variable in cond), and thus cond(p,q) is true in A, contrary to our initial assumption.
In case 2, depicted in Figure 7-2, the following must be true in state A: pseudo-path\((p,x_i)\), pseudo-path\((x_i, u)\), and link\((u,q)\); also, some entity \(y\) holds a transport right over \(x_i\); also link\((y,x_{i+1})\) is true. The transport right over \(x_i\) in \(y\) means that pseudo-link\((x_i,y)\), and thus, by transitivity, pseudopath\(( x_i,x_{i+1} )\)(via \(y\)) and pseudo-path\((p,u)\) in A. Thus cond\((p,q)\) also holds in A in this case, contrary to our initial assumption.

Thus, in either case, no single operation can result in cond\((p,q)\) if it was not true in the preceding state. Obviously then, no sequence of them can, and cond\((p,q)\) must have been true in initial state S in order to be true in \(S'\), proving that can-path\((p,q)\) implies cond\((p,q)\).

Next we shall prove that cond\((p,q)\) implies can-path\((p,q)\). We start by proving, by induction, that if \(u\) is a subject then pseudo-path\((p,u)\) implies can-path\((p,u)\). By hypothesis, we have some pseudo-path \(p =x_{i_1},x_{i_2},...,x_n = u\). Since \(u\) is a subject the pseudo-link from \(x_{n-1}\) to \(u\) \((x_n)\) must be a true link (recall that a pseudo-link is not also a link only when the potential receiver is an object holding a take right). Since a link is a path and if path is true so is can-path, can-path\((x_{n-1},u)\) is true. This constitutes the first step of our induction proof, since our induction goes backward from \(u\) along the pseudo-path.
Next we shall prove the induction step: can-path \((x_i, u)\) \(\rightarrow\) can-path \((x_{i-1}, u)\) for \(i=n-1, n-2, \ldots, 1\). There are two possible cases:

a) The pseudo-link from \(x_{i-1}\) to \(x_i\) is a true link: since a link is a path, and if a path exists then can-path is certainly true, can-path\((x_{i-1}, x_i)\) is true; then by the induction hypothesis (namely can-path\((x_i, u)\)) and the transitive properties of can-path, can-path\((x_{i-1}, u)\) is also true.

b) The pseudo-link from \(x_{i-1}\) to \(x_i\) is not a link: by the definition of pseudo-link this means that entity \(x_i\) must be an object holding \(T: x_{i-1}\). Now by the induction hypothesis we know that we can create a path from \(x_i\) to \(u\). Therefore, we may create it and move \(T: x_{i-1}\) along it from \(x_i\) to \(u\). In the ensuing protection state, since \(u\) is a subject and possesses \(T: x_{i-1}\), link\((x_{i-1}, u)\) would be true, and we would have therefore succeeded in creating a path from \(x_{i-1}\) to \(u\). By definition, then, can-path\((x_{i-1}, u)\) was true in the initial protection state.

Thus, in either case, the induction step is proven, and therefore the result that pseudo-path\((p, u)\) implies can-path\((p, u)\) (for any subject \(u\)). This result, together with the definition of cond\((p, q)\) (pseudo-path\((p, u)\) and link\((u, q)\) for some subject \(u\)) immediately yields the result that cond\((p, q)\) implies canpath\((p, q)\), the second half of this theorem.
8 Appendix B – graphPlus Source Code

8.1 AdminLib.java

package graphPlus;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

/**
 * This class implements the administrator interface to the GraphPlus access control model
 * @author Eric Brown
 */

public class AdminLib extends SimpleLibrary {

    public static void main(String[] args) {
        AdminLib lib = null;

        if (args.length > 1) {
            lib = new AdminLib(args[0], args[1]);
            lib.setFilename(args[1]);
        } else {
            lib = new AdminLib(args[0]);
        }

        System.out.println("You are node " + args[0]);
        String str = "";
        lib.usage();
        lib.display();

        while (!str.equals("exit")) {

            try {
                BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
                System.out.print("Enter a command: ");
                str = in.readLine();
                lib.process(str);
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }

    public AdminLib(String name) {
        super(name);
        if (!getNode().getType().equals(Vertex.ADMIN)) {
            throw new IllegalArgumentException("Only an Administrator can instantiate this library");
        }
    }
}
public AdminLib(String name, String model) {
    super(name, model);
    if (!getNode().getType().equals(Vertex.ADMIN)) {
        throw new IllegalArgumentException("Only an Administrator can instantiate this library");
    }
}

public void addRight(String right, String on, String to) {
    if (getNode().getType().equals(Vertex.ADMIN)) {
        Vertex onVertex = getGraph().findVertex(on);
        Vertex toVertex = getGraph().findVertex(to);
        Right rightToAdd = new Right(onVertex.getName(), right);
        getGraph().addRight(toVertex, rightToAdd);
        getDisplay().refresh();
        getDisplay().executeLayout();
    } else {
        throw new IllegalArgumentException(getNode().getName() + " is not an Administrator");
    }
}

public boolean canPath(String from, String to) {
    Vertex toVertex = getGraph().findVertex(to);
    Vertex fromVertex = getGraph().findVertex(from);
    return getGraph().canPath(fromVertex, toVertex);
}

@Override
public Vertex createNode(String name, String nodeType) {
    // Add a node to the protection system, but don't give it any rights. The admin can add these explicitly.
    Vertex v = getGraph().addNode(nodeType, name);
    getDisplay().executeLayout();
    return v;
}

public void display(ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> g, String title) {
    getGraph().DisplayGraph(g, title);
}

public void process(String str) {
    String[] commandStr = str.split(" ");
    if (commandStr[0].equals("create")) {
        createNode(commandStr[1], commandStr[2]);
    } else if (commandStr[0].equals("remove")) {
        remove(commandStr[1]);
    } else if (commandStr[0].equals("take")) {
        take(commandStr[1], commandStr[2], commandStr[3]);
    }
}
else if (commandStr[0].equals("send")) {
    send(commandStr[1], commandStr[2], commandStr[3]);
}
else if (commandStr[0].equals("display")) {
    display();
}
else if (commandStr[0].equals("addRight")) {
    addRight(commandStr[1], commandStr[2], commandStr[3]);
}
else if (commandStr[0].equals("removeRight")) {
    removeRight(commandStr[1], commandStr[2], commandStr[3]);
}
else if (commandStr[0].equals("closure")) {
    ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge>
    ClosureGraph = getGraph().canPathGraph();
    display(ClosureGraph, "Closure");
}
else if (commandStr[0].equals("path")) {
    System.out.println(canPath(commandStr[1], commandStr[2]));
}
else if (commandStr[0].equals("save")) {
    if (commandStr.length > 1) {
        save(commandStr[1]);
    } else {
        save();
    }
}
else if (commandStr[0].equals("usage")) {
    usage();
}
else if (commandStr[0].equals("exit")) {
    System.exit(0);
    // getDisplay().destroy();
}
else {
    System.out.println("No such command
");
}

@Override
public void remove(String vertex) {
    Vertex v = getGraph().findVertex(vertex);
    remove(v);
}

public void removeRight(String right, String on, String from) {
    if (getNode().getType().equals(Vertex.ADMIN)) {
        Vertex onVertex = getGraph().findVertex(on);
        Vertex fromVertex = getGraph().findVertex(from);
        Right rightToRemove = new Right(onVertex.getName(), right);
        getGraph().removeRight(fromVertex, rightToRemove);
    }
setDisplay().refresh();
} else {
    throw new IllegalArgumentException(getNode().getName() + " is not an Administrator");
}

public void usage() {
    String usageString = "Available Commands:
    " + "create nodeName type\n" + "    type = ['S' | 'O']\n" + "    remove nodeName\n" + "    take right on from\n" + "    right = ['T' | 'R' | 'Y' | 'W' | 'X']\n" + "    send right on to\n" + "    display\n" + "    addRight right on to\n" + "    removeRight right on from\n" + "    closure\n" + "    path from to\n" + "    save [filename]\n" + "    usage\n" + "    exit" + "n";
    System.out.println(usageString);
}
}
8.2 GraphAdapter.java

package graphPlus;
/* This program and the accompanying materials are dual-licensed under
* either
* (a) the terms of the GNU Lesser General Public License version 2.1
* as published by the Free Software Foundation, or (at your option) any
* later version.
* or (per the licensee's choosing)
* (b) the terms of the Eclipse Public License v1.0 as published by
* the Eclipse Foundation.
* 31 Mar 2014 - This class has been modified from its original form for use in the graphPlus research
* implementation
*/
import com.mxgraph.model.*;
import com.mxgraph.util.mxConstants;
import com.mxgraph.view.*;
import java.util.*;
import org.jgrapht.*;
import org.jgrapht.event.*;

public class GraphAdapter<V,E>
    extends mxGraph
    implements GraphListener<V, E>
{

        /**
         * The graph to be drawn. Has vertices "V" and edges "E".
         */
    private Graph<V, E> graphT;

        /**
         * Maps the JGraphT-Vertices onto JGraphX-mxICells. \{@link #cellToVertexMap\}
         * is for the opposite direction.
         */
    private HashMap<V, mxICell> vertexToCellMap = new HashMap<V, mxICell>();

        /**
         * Maps the JGraphT-Edges onto JGraphX-mxICells. \{@link #cellToEdgeMap\} is
         * for the opposite direction.
         */
    private HashMap<E, mxICell> edgeToCellMap = new HashMap<E, mxICell>();

        /**
         * Maps the JGraphX-mxICells onto JGraphT-Edges. \{@link #edgeToCellMap\} is
         */

* for the opposite direction.
*/
private HashMap<mxICell, V> cellToVertexMap = new HashMap<mxICell, V>();

/**
 * Maps the JGraphX-mxICells onto JGraphT-Vertices. {@link #vertexToCellMap}
 * is for the opposite direction.
 */
private HashMap<mxICell, E> cellToEdgeMap = new HashMap<mxICell, E>();

/**
 * Constructs and draws a new ListenableGraph. If the graph changes through
 * as ListenableGraph, the GraphAdapter will automatically add/remove the
 * new edge/vertex as it implements the GraphListener interface. Throws a
 * IllegalArgumentException if the graph is null.
 * @param graph casted to graph
 */
public GraphAdapter(ListenableGraph<V, E> graph)
{
    // call normal constructor with graph class
    this((Graph<V, E>) graph);

    graph.addGraphListener(this);
}

/**
 * Constructs and draws a new mxGraph from a jGraphT graph. Changes on the
 * jgraphT graph will not edit this mxGraph any further; use the constructor
 * with the ListenableGraph parameter instead or use this graph as a normal
 * mxGraph. Throws an IllegalArgumentException if the parameter is null.
 * @param graph is a graph
 */
public GraphAdapter(Graph<V, E> graph)
{
    super();

    // Don't accept null as jgrapht graph
    if (graph == null) {
        throw new IllegalArgumentException();
    } else {
        this.graphT = graph;
    }

    // generate the drawing
    insertJGraphT(graph);

    setAutoSizeCells(true);
}
/**
 * Returns Hashmap which maps the vertices onto their visualization
 * mxICells.
 * 
 * @return {@link #vertexToCellMap}
 */
public HashMap<V, mxICell> getVertexToCellMap()
{
    return vertexToCellMap;
}

/**
 * Returns Hashmap which maps the edges onto their visualization mxICells.
 * 
 * @return @{link #edgeToCellMap}
 */
public HashMap<E, mxICell> getEdgeToCellMap()
{
    return edgeToCellMap;
}

/**
 * Returns Hashmap which maps the visualization mxICells onto their edges.
 * 
 * @return @{link #cellToEdgeMap}
 */
public HashMap<mxICell, E> getCellToEdgeMap()
{
    return cellToEdgeMap;
}

/**
 * Returns Hashmap which maps the visualization mxICells onto their vertices.
 * 
 * @return @{link #cellToVertexMap}
 */
public HashMap<mxICell, V> getCellToVertexMap()
{
    return cellToVertexMap;
}

@Override public void vertexAdded(GraphVertexChangeEvent<V> e)
{
    addJGraphTVertex(e.getVertex());
}

@Override public void vertexRemoved(GraphVertexChangeEvent<V> e)
{
    mxICell cell = vertexToCellMap.remove(e.getVertex());
    removeCells(new Object[] { cell });
    // remove vertex from hashmaps
cellToVertexMap.remove(cell);
vertexToCellMap.remove(e.getVertex());

// remove all edges that connected to the vertex
ArrayList<E> removedEdges = new ArrayList<E>();

// first, generate a list of all edges that have to be deleted
// so we don't change the cellToEdgeMap.values by deleting while
// iterating
// we have to iterate over this because the graphT has already
// deleted the vertex and edges so we can't query what the edges were
for (E edge : cellToEdgeMap.values()) {
    if (!graphT.edgeSet().contains(edge)) {
        removedEdges.add(edge);
    }
}

// then delete all entries of the previously generated list
for (E edge : removedEdges) {
    removeEdge(edge);
}

this.refresh();

@Override public void edgeAdded(GraphEdgeChangeEvent<V, E> e) {
    addJGraphTEdge(e.getEdge());
}

@Override public void edgeRemoved(GraphEdgeChangeEvent<V, E> e) {
    removeEdge(e.getEdge());
}

/**
 * Removes a jgrapht edge and its visual representation from this graph
 * completely.
 *
 * @param edge The edge that will be removed
 */
private void removeEdge(E edge) {
    mxICell cell = edgeToCellMap.remove(edge);
    removeCells(new Object[] { cell });

    cellToEdgeMap.remove(cell);
    edgeToCellMap.remove(edge);
    this.refresh();
}

/**
 * Draws a new vertex into the graph.
 *
private void addJGraphTVertex(V vertex) {
    getModel().beginUpdate();
    try {
        // create a new JGraphX vertex at position 0
        mxICell cell =
                (mxICell) insertVertex(defaultParent, null, vertex, 0, 0, 100, 100);

        // update cell size so cell isn't "above" graph
        updateCellSize(cell);
        if (((Vertex)vertex).getType().equals(Vertex.OBJECT)) {
            model.setStyle(cell, mxConstants.STYLE_SHAPE + "=" + mxConstants.SHAPE_ELLIPSE);
        }
        if (((Vertex)vertex).getType().equals(Vertex.ADMIN)) {
            model.setStyle(cell, mxConstants.STYLE_SHAPE + "=" + mxConstants.SHAPE_HEXAGON);
        }

        // Save reference between vertex and cell
        vertexToCellMap.put(vertex, cell);
        cellToVertexMap.put(cell, vertex);
    } finally {
        getModel().endUpdate();
        this.refresh();
    }
}

private void addJGraphTEdge(E edge) {
    getModel().beginUpdate();
    try {
        // find vertices of edge
        V sourceVertex = graphT.getEdgeSource(edge);
        V targetVertex = graphT.getEdgeTarget(edge);

        // if the one of the vertices is not drawn, don't draw the edge
        if (!((vertexToCellMap.containsKey(sourceVertex) && vertexToCellMap.containsKey(targetVertex)))) {
            return;
        }

        // get mxICells
        Object sourceCell = vertexToCellMap.get(sourceVertex);
Object targetCell = vertexToCellMap.get(targetVertex);

// add edge between mxICells
mxICell cell =
    (mxICell) insertEdge(
        defaultParent,
        null,
        edge,
        sourceCell,
        targetCell);

// update cell size so cell isn't "above" graph
updateCellSize(cell);

String style = mxConstants.STYLE_NOLABEL + "=1";
if (graphT.getEdgeWeight(edge) < WeightedGraph.DEFAULT_EDGE_WEIGHT){
    style += ";" + mxConstants.STYLE_STROKECOLOR + "=red";
    style += ";" + mxConstants.STYLE_DASHED + "=true";
}
model.setStyle(cell,style);

// Save reference between vertex and cell
eedgeToCellMap.put(edge, cell);
ecellToEdgeMap.put(cell, edge);
}
finally {
    getModel().endUpdate();
    this.refresh();
}

/**
* Draws a given graph with all its vertices and edges.
* @param graph the graph to be added to the existing graph.
*/
private void insertJGraphT(Graph<V, E> graph)
{
    for (V vertex : graph.vertexSet()) {
        addJGraphTVertex(vertex);
    }

    for (E edge : graph.edgeSet()) {
        addJGraphTEdge(edge);
    }
}

8.3 GraphDisplay.java

package graphPlus;

/* -----------------------------------------------*/
* JGraphT : a free Java graph-theory library
* -----------------------------------------------

* Project Info: http://jgrapht.sourceforge.net/
* Project Creator: Barak Naveh (http://sourceforge.net/users/barak_naveh)
* 
* (C) Copyright 2003-2008, by Barak Naveh and Contributors.
* 
* This library is free software; you can redistribute it and/or modify it
* under the terms of the GNU Lesser General Public License as published by
* the Free Software Foundation; either version 2.1 of the License, or
* (at your option) any later version.
* 
* This library is distributed in the hope that it will be useful, but
* WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
* or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
* License for more details.
* 
* You should have received a copy of the GNU Lesser General Public License
* along with this library; if not, write to the Free Software Foundation,
* * Inc.,
* * 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.
* */

/** -----------------------
* GraphPractice.java
* -----------------------
* (C) Copyright 2003-2008, by Barak Naveh and Contributors.
* 
* Original Author: Barak Naveh
* Contributor(s): -
* 
* $Id: GraphPractice.java 725 2010-11-26 01:24:28Z perfecthash $
* 
* Changes
* -------
* 03-Aug-2003 : Initial revision (BN);
* 07-Nov-2003 : Adaptation to JGraph 3.0 (BN);
* 
* This class has been modified from its original form for use in the graphPlus research implementation
* */

import java.util.Map;

import javax.swing.*;

import org.jgrapht.*;
import org.jgrapht.graph.*;

import com.mxgraph.layout.hierarchical.mxHierarchicalLayout;
import com.mxgraph.model.mxIGraphModel;
import com.mxgraph.swing.mxGraphComponent;
import com.mxgraph.util.mxConstants;
import com.mxgraph.view.mxStylesheet;
/**
 * A demo applet that shows how to use JGraph to visualize JGraphT graphs.
 *
 * @author Barak Naveh
 * @since Aug 3, 2003
 */

public class GraphDisplay
    extends JApplet
{
    //~ Static fields/initializers -------------------------------------

    private static final long serialVersionUID = 3256444702936019250L;
    /**
     * An alternative starting point for this demo, to also allow running this
     * applet as an application.
     *
     * @param args ignored.
     */
    public static void main(String[] args)
    {
        // create a JGraphT graph
        ListenableGraph<Vertex, PersistentEdge> g =
            new ListenableDirectedGraph<Vertex, PersistentEdge>(
                PersistentEdge.class);

        Vertex v1 = new Vertex("v1", "S");
        Vertex v2 = new Vertex("v2", "S");
        Vertex v3 = new Vertex("v3", "O");
        Vertex v4 = new Vertex("v4", "S");

        // add some sample data (graph manipulated via JGraphT)
        g.addVertex(v1);
        g.addVertex(v2);
        g.addVertex(v3);
        g.addVertex(v4);

        g.addEdge(v1, v2);
        g.addEdge(v2, v3);
        g.addEdge(v3, v1);
        g.addEdge(v4, v3);

        GraphDisplay applet = new GraphDisplay(g);
        applet.init();

        JFrame frame = new JFrame();
        frame.getContentPane().add(applet);
        frame.setTitle("JGraphT Adapter to JGraph Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    private ListenableGraph<Vertex, PersistentEdge> graph;
//~ Constructor

// positioning via jgraphx layouts
mxCircleLayout layout = new mxCircleLayout(jgxAdapter);
mxHierarchicalLayout layout = null;

private GraphAdapter<Vertex, PersistentEdge> jgxAdapter;

//~ Methods -----------------------------------------------

public GraphDisplay(ListenableGraph<Vertex, PersistentEdge> g) {
    this.graph = g;
}

//~ Instance fields --------------------------------------

public void executeLayout() {
    layout.execute(jgxAdapter.getDefaultParent());
}

/**
 * {@inheritDoc}
 */
public void init() {
    // create a visualization using JGraphX, via an adapter
    jgxAdapter = new GraphAdapter<Vertex, PersistentEdge>(graph);
    jgxAdapter.getGraphBounds().setWidth(500);
    jgxAdapter.setLabelsVisible(true);
    mxIGraphModel model = jgxAdapter.getModel();

    mxStylesheet stylesheet = jgxAdapter.getStylesheet();
    Map<String, Object> style = stylesheet.getDefaultVertexStyle();
    style.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_RECTANGLE);
    getContentPane().add(new mxGraphComponent(jgxAdapter));
    layout = new mxHierarchicalLayout(jgxAdapter);
    executeLayout();
}

public void refresh() {
    jgxAdapter.refresh();
}
8.4 GraphMain.java

package graphPlus;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class GraphMain {

    public static void main(String[] args) throws IOException {
        SimpleLibrary lib = null;

        // args should be admin|user vertexName [filename]
        if (args.length <= 0) {
            System.out.println("usage: java -jar graph++.jar (admin | user) vertexName [filename]");
            System.exit(1);
        } else if (args[0].equals("admin")) {
            if (args.length > 2) {
                lib = new AdminLib(args[1], args[2]);
                lib.setFilename(args[2]);
            } else {
                lib = new AdminLib(args[1]);
            }
        } else if (args[0].equals("user")) {
            if (args.length > 2) {
                lib = new SimpleLibrary(args[1], args[2]);
                lib.setFilename(args[2]);
            } else {
                lib = new SimpleLibrary(args[1]);
            }
        }

        System.out.println("You are node " + args[1]);
        String str = "";
        lib.usage();
        lib.display();
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

        while (!str.equals("exit")) {

            try {
                in = new BufferedReader(new InputStreamReader(System.in));
                System.out.print("Enter a command: ");
                str = in.readLine();
                lib.process(str);
            } catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }
    }
}
public void usage()
{
    String usageString = "java -jar graphPlus.jar (admin | user) vertexName [filename]";
    System.out.println(usageString);
}

8.5 GraphModel.java

//Eric (Rick) Brown Thesis Implementation
//Dr. Victor Winter - advisor

package graphPlus;

import graphPlus.ListenableDirectedWeightedMultiGraph;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.*;
import javax.swing.JFrame;
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
import org.json.JSONTokener;

/**
 * @author Rick Brown
 */
public class GraphModel {
    private ListenableDirectedWeightedMultiGraph<Vertex,PersistentEdge> model;

    public static void main(String[] args){
        GraphModel s = null;
        if (args.length > 0){
            s = new GraphModel(args[0]);
        } else {
            s = new GraphModel();
        }

        Vertex v0 = s.addNode("S", "vertex0");
        Vertex v1 = s.addNode("S", "vertex1");
        Vertex v2 = s.addNode("O", "vertex2");
        Vertex v3 = s.addNode("S", "vertex3");
        Vertex v4 = s.addNode("O", "vertex4");
        s.addRight(v0,new Right(v1.getName(), Right.TAKE));
        s.addRight(v1, new Right(v2.getName(), Right.TAKE));
        s.addRight(v2, new Right(v3.getName(), Right.TAKE));
        s.addRight(v3, new Right(v4.getName(), Right.WRITE));
    }

    // System.out.println("Path from 1 to 0?" + s.canPath(1,0));
    // System.out.println("Path from 3 to 1?" + s.canPath(3,1));

    s.DisplayGraph(s.model, "Graph");
}
// Make a copy of the graph
ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> newGraph = s.fromJson(s.toJson());
s.canPathGraph(newGraph);
s.DisplayGraph(newGraph, "Closure");
s.persistModel();
}
/** Creates a new instance of GraphModel */
public GraphModel()
{

    // create a JGraphT graph

    model = new ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge>(
            PersistentEdge.class);
}
public GraphModel(String filename){
    this();
    try {
        FileReader reader = new FileReader(filename);
        JSONTokener jt = new JSONTokener(reader);
        JSONObject json = new JSONObject(jt);
        setModel(fromJson(json));
    } catch (FileNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (JSONException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
public ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> fromJson(JSONObject json){
    ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> newGraph =
            new ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge>(
                    PersistentEdge.class);
    JSONArray vertices = json.getJSONArray("V");
    for (int i = 0; i < vertices.length(); i++){
        JSONObject v = new JSONObject((String)vertices.get(i));
        Vertex vt = new Vertex(v.getString("name"), v.getString("type"));
        JSONArray rights = v.getJSONArray("rights");
        for (int j = 0; j < rights.length(); j++){
            JSONObject jr = new JSONObject((String)rights.get(j));
            Right r = new Right(jr.getString("vertexId"),jr.getString("privilege"));
            vt.addRight(r);
        }
        newGraph.addVertex(vt);
    }
    JSONArray edges = json.getJSONArray("E");
for (int i=0; i < edges.length(); i++) {
    JSONObject jedge = new JSONObject((String)edges.get(i));
    JSONObject jsource = new JSONObject(jedge.getString("source"));
    Vertex source = findVertex(jsource.getString("name"), newGraph);
    JSONObject jtarget = new JSONObject(jedge.getString("target"));
    Vertex target = findVertex(jtarget.getString("name"), newGraph);
    PersistentEdge edge = newGraph.addEdge(source, target);
    newGraph.setEdgeWeight(edge, jedge.getDouble("weight"));
}
return newGraph;
}

public Vertex addNode(Vertex creator, String nodeType, String name) {
    if (nodeType.equals(Vertex.ADMIN) && !creator.getType().equals(Vertex.ADMIN)) {
        throw new IllegalArgumentException("Only the Administrator can create a new Administrator");
    }
    Vertex v = addNode(nodeType, name);
    // Give the new node receive rights over its creator
    addRight(v, new Right(creator.getName(), Right.RECEIVE));
    // Give the new node's creator take and Own rights over the new node
    addRight(creator, new Right(v.getName(), Right.TAKE));
    addRight(creator, new Right(v.getName(), Right.OWN));
    return v;
}

public Vertex addNode(String nodeType, String name) {
    if (!nodeType.equals(Vertex.SUBJECT) && !nodeType.equals(Vertex.OBJECT) &&
        !nodeType.equals(Vertex.ADMIN)) {
        throw new IllegalArgumentException("New Node must be a subject or an object");
    }
    Vertex v = new Vertex(name, nodeType);
    model.addVertex(v);
    return v;
}

public boolean removeNode(Vertex v) {
    for (Vertex vert : model.vertexSet()) {
        List<Right> rightsToRemove = new ArrayList<>();
        for (Right r : vert.getRights()) {
            if (r.getRightId().equals(v.getName())) {
                rightsToRemove.add(r);
            }
        }
        if (rightsToRemove.size() > 0) {
            vert.getRights().removeAll(rightsToRemove);
        }
    }
    return model.removeVertex(v);
}
public void removeRight(Vertex v, Right right) {
    PersistentEdge edge = null;

    v.getRights().remove(right);
    // If there is an edge between these vertices, remove it and replace it with an appropriate edge
    // so we don't have duplicates
    if (model.getAllEdges(findVertex(right.getVertexId()), v).size() > 0) {
        model.removeAllEdges(findVertex(right.getVertexId()), v);
    }

    // If there is still a right in the vertex that requires an edge, add the appropriate edge.
    if (Utils.strong(findVertex(right.getVertexId()), v)) {
        edge = model.addEdge(findVertex(right.getVertexId()), v);
    } else if (Utils.weak(findVertex(right.getVertexId()), v)) {
        edge = model.addEdge(findVertex(right.getVertexId()), v);
        model.setEdgeWeight(edge, 0);
    // There is no listener for the edge weight; in order to get the visualization
    // to respond to the new edge weight, we must remove the edge and add it back again
    // so that the addEdge event will fire
    model.removeEdge(edge);
    model.addEdge(findVertex(right.getVertexId()), v, edge);
    }
}

public PersistentEdge addRight(Vertex v, Right r) {
    PersistentEdge edge = null;

    // if v is an object, adding take to it creates a pseudo-link ala Lockman & Minsky
    // for graphPlus, it is modeled as a weak edge
    v.addRight(r);

    if (model.getAllEdges(findVertex(r.getVertexId()), v).size() > 0) {
        model.removeAllEdges(findVertex(r.getVertexId()), v);
    }

    if (Utils.strong(findVertex(r.getVertexId()), v)) {
        edge = model.addEdge(findVertex(r.getVertexId()), v);
    } else if (Utils.weak(findVertex(r.getVertexId()), v)) {
        edge = model.addEdge(findVertex(r.getVertexId()), v);
        model.setEdgeWeight(edge, 0);
    // There is no listener for the edge weight; in order to get the visualization
    // to respond to the new edge weight, we must remove the edge and add it back again
    // so that the addEdge event will fire
    model.removeEdge(edge);
    model.addEdge(findVertex(r.getVertexId()), v, edge);
    }
public boolean canPath(Vertex from, Vertex to) {
    ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> pgraph = fromJson(toJson());
    canPathGraph(pgraph);

    PersistentEdge edge = pgraph.getEdge(from, to);
    if (edge != null && pgraph.getEdgeWeight(edge) > 0) {
        return true;
    }

    return false;
}

public GraphDisplay DisplayGraph(ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> g, String title) {
    GraphDisplay applet = new GraphDisplay(g);
    applet.init();

    JFrame frame = new JFrame();
    frame.getContentPane().add(applet);
    frame.setTitle(title);
    frame.pack();
    frame.setVisible(true);
    return applet;
}

public Vertex findVertex(String id) {
    for (Vertex v : getModel().vertexSet()) {
        if (v.getName().equals(id)) {
            return v;
        }
    }
    return null;
}

public Vertex findVertex(String id, ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> g) {
    for (Vertex v : g.vertexSet()) {
        if (v.getName().equals(id)) {
            return v;
        }
    }
    return null;
}

protected ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> getModel() {
    return model;
}

public ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> canPathGraph() {
    return edge;
}
ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> newGraph = fromJson(toJson());
canPathGraph(newGraph);
return newGraph;
}

public void canPathGraph(ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> g) {
    TransitiveClosureMulti.INSTANCE.closeGraph(g);
}

public void setModel(ListenableDirectedWeightedMultiGraph<Vertex, PersistentEdge> model) {
    this.model = model;
}

public JSONObject toJson() {
    JSONObject json = new JSONObject();
    JSONArray vertices = new JSONArray();
    JSONArray edges = new JSONArray();
    for (Vertex v : getModel().vertexSet()) {
        vertices.put(v.toJson());
    }
    json.put("V", vertices);
    for (PersistentEdge e : getModel().edgeSet()) {
        edges.put(e.toJson());
    }
    json.put("E", edges);
    return json;
}

public void persistModel() {
    persistModel("Model.dat");
}

public void persistModel(String filename) {
    FileWriter filew = null;
    try {
        JSONObject json = toJson();
        // System.out.println(json.toString());
        filew = new FileWriter(filename);
        json.write(filew);
        filew.close();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (JSONException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
### 8.6 `ListenableDirectedWeightedMultiGraph.java`

```java
package graphPlus;

import org.jgrapht.DirectedGraph;
import org.jgrapht.graph.DefaultListenableGraph;
import org.jgrapht.graph.DirectedWeightedMultigraph;

/**
 * a listenable simple directed graph.
 */
public class ListenableDirectedWeightedMultiGraph<V, E>
    extends DefaultListenableGraph<V, E>
    implements DirectedGraph<V, E>
{
    private static final long serialVersionUID = 1L;

    public ListenableDirectedWeightedMultiGraph(Class<E> edgeClass)
    {
        super(new DirectedWeightedMultigraph<V, E>(edgeClass));
    }
}
```

### 8.7 `PersistentEdge.java`

```java
package graphPlus;

import org.jgrapht.graph.DefaultWeightedEdge;
import org.json.JSONObject;

public class PersistentEdge extends DefaultWeightedEdge {
    private static final long serialVersionUID = 1139924853719953576L;

    public String toJson()
    {
        JSONObject json = new JSONObject();
        json.put("source", ((Vertex)getSource()).toJson());
        json.put("target", ((Vertex)getTarget()).toJson());
        json.put("weight", getWeight());

        return json.toString();
    }
}
```
8.8 Right.java

```java
package graphPlus;

import java.io.Serializable;
import java.util.Arrays;
import java.util.List;

import org.json.JSONObject;

public class Right implements Serializable {

    private static final long serialVersionUID = -5921886290286825975L;

    public static final String TAKE = "T";
    public static final String RECEIVE = "R";
    public static final String READ = "r";
    public static final String WRITE = "w";
    public static final String EXECUTE = "x";
    public static final String OWN = "o";

    public static List<String> systemPrivileges = Arrays.asList(TAKE, RECEIVE, READ, WRITE, EXECUTE, OWN);

    private String privilege;
    private String vertexId;

    public Right(String vertexId, String privilege) {
        this.vertexId = vertexId;
        if (!systemPrivileges.contains(privilege)) {
            throw new IllegalArgumentException("Right must belong to the set of valid rights");
        }
        this.privilege = privilege;
    }

    public String getPrivilege() {
        return privilege;
    }

    public String getVertexId() {
        return vertexId;
    }

    @Override
    public String toString() {
        String returnString = getPrivilege() + ":" + getVertexId();
        return returnString;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((privilege == null) ? 0 : privilege.hashCode());
        return result;
    }
}
```
result = prime * result + ((vertexId == null) ? 0 : vertexId.hashCode());
return result;
}

@Override
public boolean equals(Object obj) {
if (this == obj)
    return true;
if (obj == null)
    return false;
if (getClass() != obj.getClass())
    return false;
Right other = (Right) obj;
if (privilege == null) {
    if (other.privilege != null)
        return false;
} else if (!privilege.equals(other.privilege))
    return false;
if (vertexId == null) {
    if (other.vertexId != null)
        return false;
} else if (!vertexId.equals(other.vertexId))
    return false;
return true;
}

public String toJson() {
    JSONObject json = new JSONObject();
    json.put("vertexId", getVertexId());
    json.put("privilege", getPrivilege());
    return json.toString();
}
8.9 SimpleLibrary.java

//Eric (Rick) Brown Thesis
//Dr. Victor Winter

//SimpleLibrary.java

package graphPlus;

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class SimpleLibrary {

    private GraphModel graph;
    private Vertex node;
    private GraphDisplay display;
    private String filename;

    public SimpleLibrary(String name) {
        this(name, "Model.dat");
    }

    public SimpleLibrary(String name, String model) {
        graph = new GraphModel(model);
        node = graph.findVertex(name);
        if (node == null || node.getType().equals(Vertex.OBJECT)) {
            throw new IllegalArgumentException("Every node instantiating the library MUST do so with the name of a subject node");
        }
    }

    //Private methods to implement the logic of Take-Receive

    protected String getFilename() {
        return filename;
    }

    protected void setFilename(String filename) {
        this.filename = filename;
    }

    protected GraphModel getGraph() {
        return graph;
    }

    protected void setGraph(GraphModel graph) {
        this.graph = graph;
    }
}
protected Vertex getNode() {
    return node;
}

protected void setNode(Vertex node) {
    this.node = node;
}

protected GraphDisplay getDisplay() {
    return display;
}

protected void setDisplay(GraphDisplay display) {
    this.display = display;
}

private void take(String right, Vertex on, Vertex from, Vertex to) {
    // If the to node has the Take right over the from node and the right in question resides in the from node
    // (over the on node), then grant it to the to node.
    if (!Right.systemPrivileges.contains(right)) {
        throw new IllegalArgumentException("No such privilege");
    }
    Right takeOnFrom = new Right(from.getName(), Right.TAKE);
    Right rightToTake = new Right(on.getName(), right);
    if (to.containsRight(takeOnFrom)) {
        if (from.containsRight(rightToTake)) {
            graph.addRight(to, rightToTake);
        } else {
            throw new IllegalArgumentException("No such right in " + from.getName());
        }
    } else {
        throw new IllegalArgumentException(to.getName() + " does not have Take on " + from.getName());
    }
    display.executeLayout();
}

private void send(String right, Vertex on, Vertex from, Vertex to) {
    // If the to node has the Receive right over the from node (the calling node), and the right in question
    // resides in the from node (over the on node), then grant it to the to node.
    if (!Right.systemPrivileges.contains(right)) {
        throw new IllegalArgumentException("No such privilege");
    }
    Right receiveOnFrom = new Right(from.getName(), Right.RECEIVE);
    Right rightToSend = new Right(on.getName(), right);
    if (to.containsRight(receiveOnFrom)) {
        if (from.containsRight(rightToSend)) {
            graph.addRight(to, rightToSend);
        } else {
            throw new IllegalArgumentException("No such right in " + from.getName());
        }
    } else {
        throw new IllegalArgumentException(to.getName() + " does not have Send to " + from.getName());
    }
    display.executeLayout();
}
throw new IllegalArgumentException(to.getName() + " does not have Receive on " + from
    .getName());
    } display.executeLayout();
    }
    protected void remove(Vertex v) {
        graph.removeNode(v);
        display.executeLayout();
    }

    // Public methods - the only interface to the protection system that nodes have. Nodes can only call these, and the
    // nodeIndex with which they instantiated their Library fills in the private versions of these methods. This
    // simulates the interaction of processes in a computer system with the protection system.
    public Vertex createNode(String name, String nodeType) {
        // Add a node to the protection system and give it Receive over the creating node;
        // and give the creating node Take over the created node.
        Vertex v = graph.addNode(node, nodeType, name);
        display.executeLayout();
        display.refresh();
        return v;
    }

    public void take(String right, String on, String from) {
        String onVertex = graph.findVertex(on);
        Vertex fromVertex = graph.findVertex(from);
        take(right, onVertex, fromVertex, node);
    }

    public void send(String right, String on, String to) {
        Vertex onVertex = graph.findVertex(on);
        Vertex toVertex = graph.findVertex(to);
        send(right, onVertex, node, toVertex);
        display.refresh();
    }

    public void remove(String vertex) {
        Vertex v = graph.findVertex(vertex);
        // If this node has the Own right over v, then remove v from the model.
        Right ownOnV = new Right(v.getName(), Right.OWN);
        if (node.containsRight(ownOnV)) {
            remove(v);
        } else {
            throw new IllegalArgumentException(node.getName() + " does not own " + v.getName());
        }
    }

    public void usage() {
        String usageString = "Available Commands:
        " +
    }
" create nodeName type
" type = ["S" | "O"]
" remove nodeName
" take right on from
" right = ["T" | "R" | "t" | "r" | "w" | "x"]
" send right on to
" display
" usage
" save [filename]
" exit

System.out.println(usageString);

public void save()
    if (getFilename() != null)
        save(filename);
    else {
        getGraph().persistModel();
    }

public void save(String filename)
    getGraph().persistModel(filename);

public void process(String str)
    String[] commandStr = str.split(" ");
    if (commandStr[0].equals("create"))
        createNode(commandStr[1], commandStr[2]);
    else if (commandStr[0].equals("remove"))
        remove(commandStr[1]);
    else if (commandStr[0].equals("take"))
        take(commandStr[1], commandStr[2], commandStr[3]);
    else if (commandStr[0].equals("send"))
        send(commandStr[1], commandStr[2], commandStr[3]);
    else if (commandStr[0].equals("display"))
        display();
    else if (commandStr[0].equals("usage"))
        usage();
    else if (commandStr[0].equals("save"))
        if (commandStr.length > 1)
            save(commandStr[1]);
        else {
            save();
        }
    } else if (commandStr[0].equals("exit"))
        System.exit(0);
public void display()
{
    display = graph.DisplayGraph(graph.getModel(), "Protection State");
}

public static void main(String[] args)
{
    SimpleLibrary lib = null;

    if (args.length > 1)
    {
        lib = new SimpleLibrary(args[0], args[1]);
        lib.setFilename(args[1]);
    } else {
        lib = new SimpleLibrary(args[0]);
    }

    System.out.println("You are node " + args[0]);
    String str = "";
    lib.usage();
    lib.display();

    while (!str.equals("exit")) {
        try {
            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
            System.out.print("Enter a command: ");
            str = in.readLine();
            lib.process(str);
        } catch (Exception e) {
            System.out.println(e.toString());
        }
    }
}
package graphPlus;
import java.util.HashSet;
import java.util.Set;
public class TransitiveClosureMulti {

    //~ Static fields/initializers --------------------------------------------

    /**
     * Singleton instance.
     */
    public static final TransitiveClosureMulti INSTANCE = new TransitiveClosureMulti();

    //~ Constructors -------------------------------------------------------

    /**
     * Private Constructor.
     */
    private TransitiveClosureMulti()
    {
    }

    //~ Methods -------------------------------------------------------------

    /**
     * Computes the transitive closure of the given graph.
     *
     * @param graph - Graph to compute transitive closure for.
     */
    public <V extends Vertex, E extends PersistentEdge>
    void closeGraph(ListenableDirectedWeightedMultiGraph<V, E> graph)
    {
        Set<V> vertexSet = graph.vertexSet();
        Set<V> newEdgeTargets = new HashSet<V>();
        Set<V> newPseudoEdgeTargets = new HashSet<V>();
        // At every iteration of the outer loop, we add a path of length 1
        // between nodes that originally had a path of length 2. In the worst
        // case, we need to make floor(log |V|) + 1 iterations. We stop earlier
        // if there is no change to the output graph.
        int bound = computeBinaryLog(vertexSet.size());
        boolean done = false;
        for (int i = 0; !done & & (i < bound); ++i) {
            done = true;
            for (V v1 : vertexSet) {
                newEdgeTargets.clear();
                newPseudoEdgeTargets.clear();
            }
        }
    }
}
for (E v1OutEdge : graph.outgoingEdgesOf(v1)) {
    V v2 = graph.getEdgeTarget(v1OutEdge);
    for (E v2OutEdge : graph.outgoingEdgesOf(v2)) {
        V v3 = graph.getEdgeTarget(v2OutEdge);

        if (v1.equals(v3)) {
            // Its a simple graph, so no self loops.
            continue;
        }

        if (graph.getEdge(v1, v3) != null) {
            // There is already an edge from v1 --> v3, skip;
            continue;
        }

        if (graph.getEdgeWeight(v2OutEdge) > 0) {// tests for a strong edge
            newEdgeTargets.add(v3);
        } else if (graph.getEdgeWeight(v2OutEdge) == 0.0) {// tests for a weak edge
            newPseudoEdgeTargets.add(v3);
        }
        done = false;
    }
}

for (V v3 : newEdgeTargets) {
    graph.addEdge(v1, v3);
}
for (V v3 : newPseudoEdgeTargets) {
    E pEdge = graph.addEdge(v1, v3);
    graph.setEdgeWeight(pEdge, 0);
}

/**
 * Computes floor(log_2(n)) + 1
 */
private int computeBinaryLog(int n)
{
    assert n >= 0;

    int result = 0;
    while (n > 0) {
        n >>= 1;
        ++result;
    }

    return result;
}

8.11 Util.java
package graphPlus;

public class Utils {
    public static boolean strong(Vertex from, Vertex to) {
        // This defines the criteria for a Strong edge

        // There needs to be a Strong edge from the from node to the to node iff to has take on from and is a subject,
        // or receive on from and from is a subject.
        for (Right r : to.getRights()) {
            if (r.getVertexId().equals(from.getName()) &&
                (r.getPrivilege().equals(Right.TAKE) && Vertex.SUBJECTS.contains(to.getType())) ||
                (r.getPrivilege().equals(Right.RECEIVE) && Vertex.SUBJECTS.contains(from.getType())))
                return true;
        }
        return false;
    }

    public static boolean weak(Vertex from, Vertex to) {
        // This defines the criteria for a weak edge

        // There is an edge from the from node to the to node if to has take on from and is a subject,
        // or receive on from and from is a subject. There may be an edge from the from node to the to node if to has
        // take on from, and to is an object - hence a weak edge.
        for (Right r : to.getRights()) {
            if (r.getVertexId().equals(from.getName()) &&
                (r.getPrivilege().equals(Right.TAKE)) ||
                (r.getPrivilege().equals(Right.RECEIVE) && Vertex.SUBJECTS.contains(from.getType())))
                return true;
        }
        return false;
    }
}
8.12 Vertex.java

package graphPlus;

import java.io.Serializable;
import java.util.*;
import org.json.JSONArray;
import org.json.JSONObject;

/**
 * @author Rick Brown
 */
public class Vertex implements Serializable{

    private static final long serialVersionUID = -7418291540294294704L;
    // Static Class constants
    public static final String OBJECT="O";
    public static final String ADMIN="A";
    public static final String SUBJECT="S";
    public static final List<String> SUBJECTS= Arrays.asList(new String[]{SUBJECT,ADMIN});

    // Class members
    private List<Right> rights;
    private String name;
    private String type;

    public Vertex(String name, String type){
        this.name = name;
        this.type = type;
    }

    protected void addRight(Right right){
        // this vertex has the right on the target
        this.getRights().add(right);
    }

    public boolean containsRight(Right right){
        for (Right r : getRights()){
            if (r.equals(right)){
                return true;
            }
        }
        return false;
    }

    public List<Right> getRights() {
        if (this.rights == null){
            this.rights = new ArrayList<Right>();
        }
        return this.rights;
    }
}
public String getType() {
    return type;
}

public String getName() {
    return name;
}

@Override
public String toString() {
    String returnString = getName() + "\n";
    for (Right r: getRights()){
        returnString += r.toString() + "\n";
    }
    return returnString;
}

public String toJson(){
    JSONObject json = new JSONObject();
    json.put("name", getName());
    json.put("type", getType());
    JSONArray ja = new JSONArray();
    for (Right r: getRights()){
        ja.put(r.toJson());
    }
    json.put("rights", ja);
    return json.toString();
}

@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Vertex other = (Vertex) obj;
    if (this == obj) 
        return true;
    return true;
    if (obj == null) 
        return false;
    if (getClass() != obj.getClass()) 
        return false;
    return true;
    Vertex other = (Vertex) obj;
if (name == null) {
    if (other.name != null)
        return false;
} else if (!name.equals(other.name))
    return false;
return true;
}