INCREMENTAL VALIDATION OF POLICY-BASED SYSTEMS

ANGUS GRAHAM

A Thesis
in
The Department
Of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

May 2001

0-612-64080-9

ABSTRACT

Incremental Validation of Policy-Based Systems

Angus Graham

Policy-based systems are gaining popularity as a way to manage applications with dynamic behaviour. These systems have policies specifying the desired behaviour, entered into the system by either end-users or system administrators. In order to assure that the policies don't violate any stipulated properties of the system or conflict with one another, the policies must be validated. This validation process can take a very large amount of time as the system's policy base grows.

This thesis suggests an incremental validation method, whereby a system which has been determined to be consistent can be validated when a new rule is added to the system. "Trigger chaining" is a concept introduced in this thesis that examines which policies are triggered by the firing of a particular policy. This concept leads to new kinds of conflicts. An algorithm is suggested for incremental detection of such conflicts and is shown to operate in linear time, as opposed to complete revalidation which has quadratic complexity. Trigger chaining also leads to the detection of cyclic conflicts which are briefly discussed.

Decision tables are suggested as a suitable format for the internal representation of policies. This format provides a method of checking a policy set for completeness and could help in checking for conflicts. Also decision tables are shown to be a natural format for storing policies. It is also known how to convert decision tables into executable rules, making the transition from decision table-based policies to rule engine policies a simple one.

To my parents

Acknowledgements

The researching, preparing, and writing of my thesis has been a long, challenging, and rewarding experience. There are many people who have helped me along my journey. Some who have helped me labour over an idea, some who have graced me with their wisdom and experience, and some who have simply made my life more enjoyable. Without these people I would be no farther along than when I started. In particular, I would like to give thanks to the following people:

Dr. Thiruvengadam Radhakrishnan, for his constant flow of ideas and his inspiring words. Thanks for teaching me the art of patience and diplomacy. Thanks for giving me the freedom to explore ideas on my own and guiding me in the right direction with your wisdom.

Dr. Clifford Grossner, for helping me appreciate the skill of organization and for helping me strive for perfection. Thanks for the knowledge and insights you have given me both for the academic world and the business world.

Nortel Networks for their financial support during the researching of this thesis. Thanks to Richard Brunet for taking me under his wing while at Nortel.

Yota Karvelas for being the perfect lab partner. I have bounced many ideas off her head only to have them come back clearer and more solid. Thanks for all the laughs, and always encouraging me to turn up the music, no matter how weird it was.

My parents, Marilyn and Ray, for all the love, moral, and financial support they have given me and all the sacrifices they have made for me over the years.

Emily Bradshaw for all her love and moral support. Thanks for listening to my problems whenever I needed an ear, no matter how big, how small, or how computer science-related they may have been.

Josie McSoriley for helping me through a summer of setbacks, and for being supportive during a difficult time in my thesis work.

My cats, Daisy and Alice, for reminding me on a daily basis that above all the most important things in life are eating, sleeping, bathing, and a good chase around the backyard.

Table Of Contents

List of Figures

1. Introduction

1.1. Policy Based Systems and Policy Validation

Policy based systems offer the capabilities to dynamically change the behaviour of software. Such systems are gaining wide popularity in the industry today. Applications for these systems range from event notification software [1],[2],[3] to network management [4],[5],[17],[19] to electronic commerce [12],[11].

Policies can be entered into a system to instruct the system what actions should be taken when certain events occur, who is permitted to perform particular actions, and who is not. The system can either allow all end-users to enter their own policies, or to have one or multiple policy administrators to be in charge of this task. Policies can be entered before the execution of the system begins, but many systems allow the entry of policies to occur during system execution as well. The form in which the policies are entered varies from system to system. Some systems require that policies are entered in a strict code-like format, whereas others allow natural language input.

When policies are entered into the system, they must be checked to see if their syntax is correct. This consists of making sure the policy uses language understood by the system, and in such a way that the system understands what is meant by the policy. If the policy is not syntactically correct, the policy will not be understood by the system and there will be no way to execute it. This however, is not the only requirement a policy must meet in order to be accepted by the system.

After syntax checking the policies must be validated. When a new policy is entered into the system, it is not necessarily consistent with the rest of the system. A policy could order a combination of actions which are illegal in the application, or actions that conflict with actions specified by another policy. To detect any such anomalies, a validation process is needed. Validation can be performed at specification time, before any of the policies have been executed, or at runtime, catching conflicts as they are triggered but before they are executed.

Detecting conflicts between policies is an important concern, and solutions have been provided to tackle the problems of both specification time and run time policy validation. Validating at specification time ensures that conflicts will be detected before the execution of the system. Eliminating these conflicts before the execution of the system means that less time must be spent resolving the conflicts at runtime. Validating at runtime allows the system to catch conflicts which could not be predicted before the execution of the system. These two types of validation complement each other and therefore both forms are needed.

As policy based systems get larger, the problem of policy validation becomes more complex. Systems may have a very large number of policies, which would mean the validation process would take a long time if all policies are considered. However if a small change is made, the number of policies affected by this change could be relatively small. If the entire set of policies is validated every time a small change is made, then the system is spending a great deal of time performing validation which may not be practical, or necessary. Similarly in order to validate some systems, the policies may have to be brought offline in order to perform the validation. If the policy set is changed often, this would pose a serious problem when providing users with continuous service. Clearly this delay makes it unacceptable to revalidate the entire set of policies every time a change is made if the set of policies is large.

In this thesis, we will suggest incremental policy validation as a solution to validating large policy sets. More specifically, we will examine how to determine if a set of policies is consistent after a small change to the set has been made. This will be done by finding only those policies which are affected by the changed policies, and then validating that small subset. We will look at conflicts being detected at specification time as opposed to at run time. We will also provide a method of incremental validation that performs well as the number of policies grow. This method will be analysed to show its time complexity and compared to a non-incremental validation solution.

We will introduce the notion of storing policies in decision tables in the system. It will be shown how this format can help check a set of policies for completeness. Also we indicate that the method used for checking decision tables for consistency could possibly be used in future to detect conflicts in policies. In order to accommodate all of the policy information, the decision table format was modified. These modifications are explained, as well as the methods to enter policy in this format and to create executable rules out of this format.

The concept of "trigger chaining" has been introduced in this thesis in order to see which policies will be triggered by the firing of a particular policy. This new concept introduces a new kind of conflict, in that a policy firing immediately after another may undo the actions of the first. This new type of conflict is explained and a method of incremental validation is developed in order to detect these conflicts. The method is analysed and compared to performing an exhaustive revalidation of the entire system. The concept of trigger chaining also introduces another kind of conflict, a cyclic conflict, which is introduced but not discussed in detail in this thesis.

Policy scope is something we have introduced into our policy model. Scope is what nodes of the system a particular policy should affect. Some policies should affect the whole system, while other policies may only be applicable to certain areas of the system. Scope is an important factor when looking at policy conflicts. Two policies which might otherwise conflict will not if their scopes don't overlap.

1.2. Organization of this Thesis

In Chapter 2 policy and its terminology are introduced, and a couple of policy frameworks are presented as examples. We also present the policy model used in this thesis. Chapter 3 introduces the benefits of storing policies in a modified decision table format. In Chapter 4 we present a solution to the problem of incremental policy validation. We also introduce the concept of trigger chaining and provide an incremental validation solution for trigger chaining conflicts. An analysis of the algorithm is performed in this chapter to demonstrate that the algorithm performs well as the number of policies gets to be very large. In Chapter 5 we present an existing policy framework and demonstrate how our incremental validation technique would be added to this system. Chapter 6 examines the contributions of this thesis, while Chapter 7 discusses directions for possible future work.

2. Policy Based Systems

Policy-based systems have been studied by many different researchers in recent years. This chapter gives a summary of their research. We will start by giving a quick introduction to policies and the common terminology used when discussing policies. Next we will discuss why there is so much interest in policy-based systems and how policies can benefit software. We will next look at various policy models, in particular looking at how they specify policies, detect and resolve policy conflicts, and perform the execution of the policies. Finally, we will discuss our own policy model which we have developed using earlier models as a base. Our model will also introduce a new concept: policy scope, which is not present in any earlier models.

2.1. Introduction to Policies

The definition of **a policy** given by Lupu and Sloman [4] is "information which can be used to modify the behaviour of a system". A policy is often made up of a set of rules related to a particular aspect of an entity. Use of policies is not restricted to software. They can be used to describe how situations are to be handled in many real-life scenarios. An example would be a professor's policy governing the submission of a term paper. In this case the policy is made up of rules which the professor follows to determine what to do when a paper is handed in late, etc.

A **rule** is defined as a set of actions, which are either to be performed on, or prevented from executing on a particular entity when certain criteria are met. The rules have an attribute specifying which events trigger the rule. However, there is also a rule attribute called a constraint. The constraint is a second-level condition that must be matched in order for the rule to fire. Rules following this format are said to follow the Event Condition Action (ECA) rule paradigm [12]. Several systems use the ECA rule paradigm [17],[4],[12]. The object which is to be manipulated by the actions of the policy is called the **policy target** whereas the object which is interpreting the policy is the **policy subject** [6].

Because there are multiple policies in a given system, it is important to examine how one policy will affect another. If there are two policies which are triggered to fire at the same time, and they contradict one another, these policies are said to **conflict**. By contradict, we mean that one policy states to perform an action a, and another policy states to perform the opposite action $\bar{a}$ or an incompatible action b. A pair of incompatible actions would be two actions which cannot be performed by the same subject at the same time, for example proposing a budget and approving a budget. Cholvy and Cuppens [7] say a rule is **inconsistent** if there exists a world such that it leads to a conflict. We can therefore say that a rule which does not lead to a conflict is **consistent**.

Intuitively a conflict refers to a **real conflict**. This means should the system execute, the conflict will definitely occur and interrupt the system's execution if no attempt at resolving the conflict is taken. The process of checking policies to see if they conflict is called **policy validation**. The term validation in this context has a different meaning than the one associated with software engineering. The term validation in software engineering is the process of checking if the implemented software meets the expectations of the user [8]. Although policies could be checked to see if they represent the expectations of the user, the term validation will not be used to this end.

Often times, conflicts can be detected when the policies are entered into the system [9]. It is very important to deal with conflicts that are detected at specification time, because when the system executes, the conflict will definitely occur, resulting in an execution problem. This could be due to a specification error, and so it is important to detect any possible conflicts at specification time and notify the administrator [6].

Conflicts can also be detected at run-time. Although it is often advantageous to detect potential conflicts at specification time, this may not always be possible. Some conditions are based on states of the system which are unknown at the time of policy specification [6],[9]. In this case, the only way to detect the conflict is while the system is executing [4]. Although the notion of run-time conflicts is an important area of conflict detection in policy systems, this thesis will focus on the problem of specification time conflict detection.

A **potential conflict** is present in a set of policies when there are two policies which may or may not result in a conflict at runtime. The conflict is dependent on some variable values or the state of the system. The difference between a conflict and a potential conflict is that a conflict will definitely occur at runtime whereas a potential conflict may or may not happen at runtime. For example, if one rule fires when $x = 5$ and another rule with opposing actions fires when $x = y$, it is impossible to determine whether the two rules will be triggered at the same time before the system is running and the value of y is known. If a system has a potential conflict, and at runtime the state necessary to trigger the potential conflict is reached, then the potential conflict becomes a real conflict [10]. In the example given above, if at runtime we see that the value of y is 5, then the two rules will trigger at the same time and we get a real conflict.

It is important to look at Cholvy and Cuppens' definition of an inconsistent system again [7]. They state that if there exists a world such that there is a conflict in that world, then the system is inconsistent. The world they refer to would be the state of the system, including the state of all attributes in the system, which events are occurring, etc. Therefore even if there is a potential conflict, the system is considered inconsistent according to Cholvy and Cuppens. If a system has a real conflict in it, it too then would be considered inconsistent, but to indicate the difference, we will say that the system is in a state of conflict.

Definition 2.1.1:

A system is said to be inconsistent if and only if there is a potential conflict or a real conflict between two or more of its policies.

When a system is in a state of conflict, it is important to somehow resolve the conflict. In order to resolve the conflict, one or more actions are cancelled, but this is not an obvious solution. Actions cannot be cancelled arbitrarily, and the goal is to remove the least number of actions. There are two philosophies to canceling actions. If all actions are assumed to be atomic then canceling one action will not affect the others. The other approach is to cancel all actions that are triggered by the same event. In this approach either all the actions associated with a particular event will succeed together or fail together. This, although it cancels more actions, sometimes makes more sense. For example if a rule states to perform three actions which are all related to each other, such as *package item*, *update inventory*, and *ship item*, and the action *package item* is canceled, then the entire process will no longer make sense. It would make more sense to cancel all three actions in this case [12].

2.2. Why Policies are Needed

In today's world, software users are no longer content with software that performs a fixed set of functions in a preset manner. Users have various individual needs, and they want software to meet those needs. In recent years the software industry has been moving towards building software which can be customized by the user so that it can meet the individual's needs. Policies are one way in which this customizability can be delivered. By developing a common policy framework, software can easily be built around this framework, without having to design a new method of making the software customizable for each software system.

Policies also separate the behaviour aspect of the software from the main functions. This allows either the main functionality of the software, or the user's custom behaviour to be changed without affecting the other aspects of the software. This is very important, since one would not want a user to have to redefine all of their rules if one of the non-customizable components of the system was upgraded. Similarly one would not want to have to change the code of the main software system if the policy framework is upgraded. For this reason, policies are interpreted as opposed to being compiled [4].

Policies also offer a level of abstraction from code. If rules had to be defined in code, then customizing the software would be a very tedious process. An easy to use front end interface is a solution to this, but in the end, all custom behaviour needs to be defined in a low level form which can be understood by the software. If, say, the user interface lets users enter their behaviour in natural language, then there may be vague terms or double meanings which may prevent analysis of the policies at that level. Similarly analyzing it at a low-level such as code level is not always ideal either, since at that level abstract concepts like what the entire series of given commands is trying to achieve is lost. There should be some middle layer which is not as abstract as natural language and does not have any of its ambiguities, but at the same time still carries the overall concepts and goals which the low-level code loses. Policies offer this middle layer. Users can enter policies directly using some formal policy definition language, or can enter it at a higher level and have it translated to the definition language automatically. A policy can be analyzed for correctness and consistency before it is translated to the final code to be executed [17].

2.3. Policy Models

In this section we present various policy models created by other researchers. We discuss the features of each model and indicate both their strengths and weaknesses. The main criteria being examined are how the policies are entered into the system and how conflicts are detected and resolved, and how policies are executed. Finally two complete policy frameworks are presented.

2.3.1. Specifying Policy

In order for policy based systems to get their policies, there needs to be some way a policy administrator can enter policies into the system. The system may have only one policy administrator, or it may allow all of its end users to enter policies. The system could even have multiple levels of policy administrators. For example, a manager may be permitted to set policy for all of his subordinates, while the subordinates can only set policies for themselves.

Sometimes policy is specified at a fine level using mathematical formalisms such as first order logic [13],[14], sometimes it is stated at a very high-level such as natural language [17], or a GUI based interface [15]. More commonly, however, a **policy definition language** is used to specify the policy. Even in the case that policy is specified at a higher level, the policies are often converted into a policy definition language form, which can then be used to analyze them [17],[14].

The role of the policy definition language is to represent the conditions that trigger the policy and the actions to be carried out when it is triggered. It must be formal enough to eliminate any possible ambiguities found in natural language. It should also be high-level enough to be able to represent the concept of the policy's behaviour as opposed to a series of low-level instructions [13]. This will allow the policy to be analyzed for such things as correctness of behaviour before it is translated to low-level instructions. Not only this, but being at a higher level than code will make it easier for the administrator to enter the policies, and understand them when re-reading them in the case that he needs to make further changes to the policies.

Jajodia, Samarati, and Subrahmanian [16] proposed the Authorization Specification Language (ASL) in order to specify access control policies. It provides **authorization** policies to specify whether an action is authorized or denied for a particular combination of [<object>, <user>, <role>] sets. Since the language is meant for access control purposes, this single type of policies is sufficient. One problem with this model, however, is that all authorizations are given the same level of importance.

Bertino, Jajodia, and Samarati [21] proposed a model that is also meant for authorization policies only. They do, however, divide the notion of authorizations into two levels: weak authorizations and strong authorizations. If there is a strong authorization for one action and a weak authorization for the opposite action, the strong authorization always overrides the weak one. When there is a strong authorization for one action and another strong authorization for the opposite action an inconsistency is said to have occurred. This model was used for access control policies for databases, so having only authorization policies was sufficient. This may not be flexible enough, however, for other types of applications. There are other languages, which provide further types of policies, such as a policy enforcing that an action must be done.

Koch, Kress, and Krämer [17] proposed a language where policy can have three possible modality values: obligation (having to do something), permission (permitted to do something), and prohibition (not permitted to do something). These three modality values take their base from standard deontic logic [27]. This seems to be a common approach that is used, with some variation, in other models [1][18][19]. The authorization policies used by Jajodia, Samarati, and Subrahmanian are similar to policies in Koch, Kress, and Krämer's language using the permission and prohibition modality values.

Another language proposed is that of Lupu and Sloman [4]. In their language they use only two modalities: authorization and obligation. They also allow positive and negative modifiers to be used with these modalities, so that rules can have positive or negative authorization (can or cannot do), or positive or negative obligation (must or must not do).

Lupu and Sloman state that their notion of authorization is independent of their concept of obligation, and so their approach is not based on deontic logic as Koch, Krell, and Krämer's is. According to Lupu and Sloman, whether it is permissible to do something or not has no bearing on whether or not that action must be done. With Lupu and Sloman's representation, both of these states can be represented without affecting the other. E.g., a person may have permission from the file system to delete all his or her files, but that person's boss may state that they must not do it. In this case the person has positive authorization to delete his files but also has negative obligation for the same action.

One of the main differences lies in where the modality is interpreted. For obligations the modality is interpreted at the subject, whereas authorizations tend to be interpreted at the target. This is because authorizations are designed to protect the target, so subjects cannot be trusted to obey the modality in this case. On the other hand, if there is an obligation such as "The subject must not disclose any information to the target until after the final submission date", the target may in fact want to receive the information contrary to the rule. Therefore in this case it is the subject which must interpret the modality in order to ensure that it is taken into account [4].

Cuppens and Saurel's language [20] provides each rule with a more powerful conditional argument. Instead of only specifying a requirement that must be met in order for the rule to fire, their language also provides the keywords *before*, *during*, and *after* for use in the conditional argument. This means that policy designers can specify that the policy should fire before, during, or after a certain condition has been met.

2.3.2. Detecting and Resolving Conflicts

Lupu and Sloman define two types of conflicts that can occur between policies given the above model: **modality conflicts** and **application specific conflicts.** According to their definition, a modality conflict arises when two policies with opposite modality refer to the same subject, actions, and targets. This can happen in three ways:

- The subjects are both obligated-to and obligated-not-to perform actions on the targets.
- The subjects are both authorized and forbidden to perform actions on the targets.
- The subjects are obligated but forbidden to perform actions on the targets.

Note that since Lupu and Sloman do not imply authorization with obligation, the third type of conlfict is possible [6].

Application specific conflicts occur when two rules contradict each other due to the context of the application. Examples of this are:

- conflict of priorities for resources
- conflict of duties
- conflict of interest
- actions that perform opposing tasks

For example, assume that one rule instructs user X to submit a budget to the organization he works for, and say that another rule instructs the user to approve all budgets in the system. From looking at the policies without any knowledge of the world there is no apparent conflict, but with common sense, we know that in most organizations the person who approves budgets cannot approve his own budget. This of course may or may not be relevant to a particular application. Therefore, these restrictions are unique to the application and the rules cannot be considered to conflict without some sort of extra application specific knowledge. This extra knowledge is entered in the form of meta-policies. The meta-policies describe assumptions the system must make and explains how the system's "world" works. A set of policies may be consistent in one application and inconsistent in another due to the difference in the applications' meta-policy or world rules [6].

Lupu and Sloman provide a method of checking policies as they are specified for conflicts. Although they acknowledge the need for run-time conflict checking, they do not discuss any methods which would be useful in performing this type of checking. Similarly, their method of static (specification time) policy validation involves examining the entire set of policies, or at very least all the policies found in one domain. They do not provide any method of incrementally checking a very small number of policies when a new policy is added or modified, to determine if the new set of policies is consistent [6].

Fraser and Badger [23] proposed a way to detect conflicts for new rules that are introduced at run-time. Their approach examines the new rules to see first if the new policy is well formed. If so the next test is to see if it conflicts with the rest of the rule base, and if so, the new rule is rejected. The technique proposed, however, was for use with the Domain and Type Enforcement prototype kernel, and so was specific to DTE.

Howard, Lutfiyya, Katchabaw, and Bauer [22] provide a generic policy based architecture. Their approach is designed to allow dynamic modification of policies at run-time. They do not, however, discuss the notion of policies conflicting with each other in any way, and therefore do not provide any mechanisms to check for conflicts. They do provide a policy validation service in their architecture, but its purpose is to check if any managed objects violate any policy rules.

Chomicki, Lobo, and Naqvi [12] propose a way to resolve conflicts. Instead of detecting conflicts at specification time, they perform detection and resolution at run time. There are monitors in the system which detect actions that cannot occur together, and then decide whether to delete one of them, or delete one of them plus the other actions that were meant to execute along with it. They suggest obtaining priorities for each policy from the user in order to decide which actions should be canceled when conflicts arise. Since the detection and resolution is performed at run time, predicting which rules will conflict is not a problem. However, there may be some rules which by examining them at specification time, we know will always conflict. If these conflicts were detected at specification time, the policy administrator could modify the rules so that this conflict is avoided at run time. Their approach offers no specification time conflict detection, which could eliminate some of the automatic conflict resolution needed.

Lupu and Sloman also propose a few ways to resolve conflicts once they are detected. The first is to always give priority to negative policies. For example if one policy gives positive authorization for an action and another policy gives negative authorization for the same action, the policy giving negative authorization overrides the positive one. The second method of resolution they suggest is to give each policy an explicit priority value. This can then be used to give rules precedence in the event of a conflict. The third method they suggest is to examine the distance in the class hierarchy of the object being managed from the policy managing it. For example, let us say one policy is from the person class and another conflicting policy is from the human resources employee class which is a subclass of the employee class, which is a subclass of the person class. In this case the latter rule takes precedence, since the managed object, the human resources employee, is closer to that subclass than the general person class in the class hierarchy. The final method suggested is similar to the above method, but it is based on the domain structure, as opposed to the class hierarchy. A policy applying to a subdomain takes precedence over a policy applying to an ancestor domain because it is more specialized and targets fewer objects [6].

2.3.3. Policy Execution

Moore, Ellesson, Strassner, and Westerinen [26] state that a declarative approach to policy systems is better suited than a procedural one. They do not rule out the possibility that a policy framework can be successfully implemented using a procedural approach, but they think it would not be as natural an adaptation. This allows various implementations to use different algorithms for the testing of conditions and the sequence of action execution. The declarative rules state only the relationships between attributes to be tested and the actions to be executed. The opinion that a declarative approach is more suited to the problem of policy systems appears to be a common one, as many other researchers propose policy frameworks in the same manner [4][17][12].

2.3.4. Policy Implementations

2.3.4.1. IETF's Policy Framework

The Internet Engineering Task Force (IETF) has been developing a policy framework that is vendor independent, interoperable, and scalable. Although they deal mainly with the application of network management, they acknowledge that their policy framework could easily be adapted for other uses as well.

There are three main requirements according to the IETF that a policy framework must meet. First there must be a repository in order to store and retrieve policies. By storing all the policies in one location, the policies can be re-used by several devices. This also helps maintain consistency throughout the policies [24].

Second, there must be a common format to enter the policies. With a common policy format that is vendor independent, the same policies can be compatible with many different applications. For example, a management application may need to manage several similar devices, such as printers, that come from different vendors. If there is a standard policy format being used, then the same policy can be used to manage all the devices. This prevents the policies for each device from having inconsistencies as they might if the same policy was entered for each device in a slightly different format [24].

Finally, a policy framework needs a way to communicate the policy beyond the repository. This is necessary for scalability [24]. If there are a large number of nodes accessing the repository for policies, a bottleneck will occur. There needs to be some way to distribute the policies so that the repository is not overwhelmed with requests for policy information.

In addition to these three main requirements, they have identified a few other issues which should be examined: security, timely delivery of policies, and dealing with mobile devices [24]. We will not dwell on these as they are beyond the scope of this thesis.

IETF's approach differs from management tools such as SNMP [25] in that they propose some degree of automation of management. When the system already contains information that is needed to manage the system, the policy can retrieve it automatically without needing an administrator to re-enter it. This helps achieve the goal of managing the network with the minimum amount of human intervention [24].

In addition to the requirements of the framework itself, the IETF have identified certain requirements that must be met by a policy specification language (PSL):

- a way to state which actions should be taken by the policy-managed entity
- a way to state which conditions must be met in order to take the above actions
- a way to state what the policy is to control (policy subject)
- getting status information about the policy subject
- describing properties of the policy subject
- describing the relationship of multiple policy actions (e.g., one is dependent on the completion another, etc.)
- security information

In IETF's model, policy is represented by a set of policy rules. Each rule is made up of conditions and actions. The rules can be grouped into policy groups. These groups can be nested, and therefore give a way to represent policy hierarchy. A policy group is limited to an aggregation of policy rules or other policy groups. A policy group cannot be both. Policies do not need to be contained in a policy group. A one-rule policy that is not part of a group is called a stand-alone policy [26].

A rule's set of conditions can either consist of an ANDed set of conditions (Disjunctive Normal Form) or an ORed set of conditions (Conjunctive Normal Form). When a rule's set of conditions evaluate to TRUE, the actions associated with that rule are executed. The rule can specify to execute the actions in a particular order, state that the order is mandatory or merely recommended, or state that there is no recommended order of execution at all [26].

IETF also allows their rules to have a priority value. Two policies may apply at the same time but may state different things. This is where priority values come in handy. For example, one policy may state that everyone in the department gets 100 pages of printer quota, but another policy may state that Dr. Jones gets 500 pages of printer quota. Since both rules apply, there needs to be some way to distinguish which one should take priority. In this case the rule stating that Dr. Jones gets 500 pages of printer quota should have a higher priority to indicate that it takes precedence over the other rule [26].

IETF's policy model was designed with a declarative, non-procedural approach. In declarative languages, rules and functions are declared and then executed according to an execution algorithm. The rules are not specified to execute in a particular order. This contrasts to a procedural approach where the order of execution of various functions is strictly stated. Although their model can be implemented using a procedural language, it is better suited to declarative languages, since the condition testing sequence and action execution sequence are not specified in the policy repository [26].

IETF's framework uses roles which policies are associated with, instead of assigning policies to specific resources. The policies are associated with the roles and the resources are also associated with various roles. If the role of a resource changes, there is no need to change its policies. All that needs to be done is to reassign the resource to a different role which has a different set of policies associated with it. Also if the policies governing a certain role are changed, then all resources associated with that role will take on the new behaviour. Compare this to having to change the policies for each of the individual resources, possibly introducing inconsistencies among the various policies due to human error [26].

Roles also help in avoiding certain conflicts. By specifying certain policies for one role and other policies for another role, all the policies can coexist even if some of the policies conflict. As long as all the rules in each rule are consistent, and the two roles cannot be active at the same time, there will not be a conflict. This can make specifying policies easier, since not all policies need to be consistent with each other when there is no chance they will be used at the same time [26].

2.3.4.2. Lupu and Sloman's Policy Framework

Lupu and Sloman proposed a management framework that included a policy service. Their framework consists of:

- A domain service to group target objects.
- A policy service to allow policies to be specified, stored, and distributed to agents which will interpret them.
- A monitoring service to monitor managed objects for occurring events and to notify interested agents [4].

A domain is used to group objects based on certain attributes, such as geographical location, object type, functionality, etc. or simply even for the user's convenience. In Lupu and Sloman's framework, domains contain multiple objects and can also contain other domains, called subdomains. Domains can also overlap with each other. This means that an object X can be a member of two or more domains. Path names are used to identify domains. For example, domain C which is a subdomain of B which is a subdomain of A can be referred to as /A/B/C [4].

The policy service is the component of the system that allows a user to specify the behaviour of their policy. The service provides tools to the user to define policies. All the policies are also stored in this component as objects. Policies can be members of domains. This allows the policies to have other authorization policies governing who can access and modify them [4].

A policy editor is used by an administrator to enter, modify, delete, enable, and disable policies in the policy system. The administrator checks for any conflicts and then changes the policies accordingly. Authorization policies are then distributed to target agents, and obligation policies are sent to subject agents [4]. Once the policies have been distributed to the appropriate targets and subjects, they may be enabled or disabled by the administrator without having to redistribute the policies [28]. There are manager agents which register themselves with the monitoring service. This allows the manager agents to receive any relevant events coming from the managed objects. If a manager agent receives an event which triggers an obligation policy, the agent queries the domain service to determine which target objects are affected, and then performs the actions stated in the policy on the targets [4].

The policies in the system are represented by rules. The rules have an attribute specifying which events trigger the rule. However, there is also a rule attribute called a constraint. The constraint is a second-level condition that must be matched in order for the rule to fire. For example a rule may fire upon receiving event A but only between the hours of 9:00am and 5:00pm. This time period condition would be expressed in the constraint part of the rule. As with most rules there is also an action attribute stating which actions should be taken upon the rule firing [28].

Roles associate behaviour with a position as opposed to a particular individual. For example if Eric Jones is a manager, he has certain responsibilities based on that manager position. If he were to be promoted to senior manager, then his responsibilities would change. If we were to modify his policies to reflect his new position, we may introduce errors, or the responsibilities tied to his position may not be consistent with someone else's policies who is in the same position. It is much simpler to associate those responsibilities with the roles of manager, and senior manager themselves. Then the individual can be associated with the role. If the role changes, the individual can simply be associated with a new role and disassociated from the old one [6].

Lupu and Sloman also suggest that it is useful for each role to have its own context in which to operate. This prevents a user from playing multiple roles at once, and performing actions in one role which he is not permitted to but is permitted by another role. For example, if Role A permits an action and Role B prohibits it, then the behaviour would be determined by whichever role is active at the time [6]. A good example of this is a police officer. While on duty a police officer has certain privileges such as to drive faster than the speed limit, and certain restrictions such as not being able to consume alcohol. Off duty the police officer does not have all of the same privileges or restrictions. His role of being a police officer is no longer active and so the privileges and restrictions associated with that role no longer affect him.

Roles are implemented using domains. The positions of the organization to be modeled can be represented by domains. A role is then considered to be the set of policies with a particular domain as the subject. A person, represented as an object, can be added to or removed from the domain, without changing the policies associated directly with that person, or the role the person takes on [4].

Roles often have relationships, which are ties between two roles. For example, a senior manager has a relationship with each of its managers. Lupu and Sloman also provide a way to associate these relationships to the roles using policies. This allows policies to express the relationship between roles in terms of rights and duties of both parties towards each other [6].

This framework could easily be generalized to serve as a general purpose policy application framework. Domains can still be used to group objects together, and roles can still be used to show relationships and responsibilities between objects.

2.4. Our Policy Model

In this thesis, we are attempting solve the problem of incremental policy validation regardless of what application the policy is being used for. We have created our own policy model, which takes some of its elements from work by other researchers. In particular policy specification and policy execution are taken straight from other work. We have based our conflict detection on other work, but have extended it to be more efficient and more complete. Also, we have introduced a new concept into our model: policy scope.

2.4.1. Policy Specification

Out of all the specification languages we examined, Lupu and Sloman's language appears to be the most flexible. It offers both positive and negative modifications of authorization and obligation policies, and allows policies to have priority values. We find that it meets all the needs for policy specification. Therefore we have chosen to use their policy specification language for our policy model. Although this thesis does not focus on policy entry, we assume that all policies entered into the system are done so in the form of Lupu and Sloman's policy specification language.

2.4.2. Detecting and Resolving Conflicts

Since we are using Lupu and Sloman's policy specification language, our policies can have the same type of modality and application specific conflicts indicated by Lupu and Sloman. Therefore our conflict detection methods will aim to find these types of conflicts. Lupu and Sloman provided ways to detect these conflicts by examining the policies at specification time. In this thesis we will provide a way to determine if a set of policies is consistent when adding a new rule to a pool of consistent rules. We also include some new types of conflicts. If we know that one policy firing will cause another policy to fire, we introduce new conflicts that we can detect. We call this type of conflict a trigger chaining conflict and we will discuss it further in Chapter 4.

2.4.3. Policy Execution

Our model will store policies in an internal policy format in order to perform conflict checking and other analysis on the policies. In order to execute the policies, they must be transformed from this internal representation to a format which can be executed. Although we are not concerned with how this is done, for the purposes of this thesis we assume that after conflict checking and analysis, the policies are transformed into rules that can be entered in a rule engine such as an expert system. There has already been a lot of research done in the field of rule engines and many rule engines are available on the market today.

2.4.4. Policy Scope

In our model we introduce the concept of policy scope. Up to this point it has been implicitly assumed that all policies apply to the entire system. This, however, may not be desirable. It may be beneficial to have some policies, which, rather than affect the system as a whole, only affect a well-defined subset of the system. In this case we call policies which do affect the whole system **global policies**, and policies which are local to a particular subsection of the system **local policies**. Take for example an organizational hierarchy. There may be an administrator who is authorized to set policy for the entire organization. In this case his policies would be global. On the other hand, there may be an individual user who wants to set a policy for how his e-mail should be delivered. In this case he should not be able to set the policy for everyone, only for himself. In this case it would be a local policy. Note that in some cases the policy could be global or local depending on the scope of the hierarchy being examined. For example if a manager is able to set a policy for all his subordinates, the policy would be considered global to the department, but it does not affect the entire organization. Therefore, when looking at the entire organization, the policy would be considered local.

An example of this would be a university department that has policies governing printers and how they are used. The department owns the printers so every member of the department uses the same policy for all the printers, therefore it can be considered a global policy. If a professor, however, purchases his own printer for use by his students, he may have his own custom policy he wants to enforce over his printer. In this case the printer policy for the private printer is not applicable to the whole system and is a local policy. Note that because global policies reach to the same nodes that are governed by local policies, local policies must have a higher priority value. Without this higher priority the local policy will never be taken into account since the global policy will be used all the time.

Similarly, the concept of consistency can be divided into two subdivisions: **globally consistent** and **locally consistent**. If a system is completely consistent, that is to say that every single one of its policies coexist with all other policies in the system without causing conflicts, then we can say that the system is globally consistent. However, we may be able to predict that there may be little chance of a particular policy ever interacting with another policy. In this case it may be unnecessary to strive for global consistency. Instead it may be necessary to determine only that certain subsets of policies are consistent within themselves. If a subset's policies can coexist with all other policies in the subset without causing conflict, but may cause conflicts with policies not in the subset, we say that the subset is locally consistent and globally inconsistent. A policy that is globally consistent implies that it is locally consistent. Similarly a policy that is locally inconsistent implies that it is globally inconsistent.

Theorem 1:

Let S be the entire set of policies, and let there be a set of policies X such that $X \subseteq S$. Then we can say:

1.1. GloballyConsistent(S) $\Rightarrow$ LocallyConsistent(X)

1.2. LocallyConsistent(X) $\not\Rightarrow$ GloballyConsistent(S)

1.3. LocallyInconsistent(S) $\Rightarrow$ GloballyInconsistent(X) [Dual of 1.1]

1.4. GloballyInconsistent(X) $\not\Rightarrow$ LocallyInconsistent(S) [Dual of 1.2]

Proof:

1.1. If S is globally consistent then every policy in S can coexist with every other policy in S without causing any conflicts. Since all policies in X are policies in S, then every policy in X can coexist with every other policy in X without conflict. Therefore X is locally consistent.

1.2. Say we have a set of policies X such that every policy in X can coexist with every other policy in X without causing conflicts. Then X is consistent. Let us say that one policy in X says "Close door A at 1:00pm". Now say that S is made up of all the policies in X plus one more policy which says "Open door A at 1:00pm". Since this new policy conflicts with a policy in X, we must in this case say that S in inconsistent. Therefore X being consistent does not imply that S is consistent.

1.3. If X is inconsistent then there are two or more policies in X that conflict with each other. Since S contains all the policies in X, it also contains the policies which conflict. Therefore S is also inconsistent.

1.4. Say that the set of policies S contains two policies: "Open door A at 1:00pm" and "Close door A at 1:00pm". These policies conflict and so S can be said to be inconsistent. Now say that we have X which is made up of only one of these policies. Since X contains only one policy, it cannot be inconsistent with any other policies. Therefore global inconsistency of S does not imply local inconsistency of S.

A local policy is not compared at every node when checking for global consistency, since the scope of local policies does not reach the entire set of nodes. Instead it is only examined for local consistency at the nodes it affects. If a local policy is inconsistent at any node it affects, then it is impossible for the system to be globally consistent. One possible way of resolving a global conflict is for the policy administrator to make the conflicting global policies each local to a particular node or set of nodes. If only one of the policies is made local then one of the policies is still global and there will still be a conflict in those selected nodes that are locally affected by the first policy. For example say that global policies A and B conflict. Now say that in order to fix the conflict, policy A is made local to node N1. Now there will be consistency on every node, except for N1, because B being global, it too affects N1. Therefore N1 is still locally inconsistent. Similarly, if policies A and B were both made to be local to N1, then N1 would of course be locally inconsistent, thereby not changing the status of the system from being globally inconsistent.

Since we know that a system containing nodes that are locally inconsistent is globally inconsistent, we can use this information when checking for the consistency of the system. If one node is found to be locally inconsistent, we can halt validation and declare that the entire system is globally inconsistent, without having to check the remaining nodes.

3. Policy as Decision Tables

Decision tables are a formal way to specify the behaviour of a software system. It groups conditions with actions that will be performed upon the conditions being met. These groups of conditions and actions are called rules and stored as a column in the table. Decision tables offer an easy way to analyze whether a set of rules is complete and consistent. This will benefit us by allowing policy rules to be checked for completeness, as discussed in section 3.2.1.

3.1. Introduction to Decision Tables

A decision table is made up of four parts. The top left quadrant of a decision table is called the condition stub. The condition stub lists all the possible conditions that can occur in the system. It can be listed as asking a binary question (yes/no, do/don't do, etc.) or something more general such as colour or value. The bottom left quadrant is called the action stub. The action stub lists all the possible actions that can be performed by the system. As with the condition stub, the items in the action stub can be written to expect a binary answer (yes/no for example), or to expect a value such as "Sell car for", which expects a dollar value [29].

Michael Montalbano defines the condition stub as a vector of conditions and an action stub as a vector of actions. He then defines a condition as a set of condition values and an action as a set of action values [29]. I have extended these last two definitions in order to make them more complete. According to Montalbano, a condition is a set of condition values. Therefore a condition called *Cost* whose value set is the set of integers and another condition called *Sold for* whose value set is also the set of integers would be identical, since their values are both from the set of integers. I find that in order for the condition to have any meaning, it must be given some sort of context. The clue for that context is often contained in the name of the condition. Therefore when I refer to the term *condition* I will be referring to a set of condition values and the condition name associated with that set. Similarly, when I refer to an *action* I will be referring to a set of action values and the action name associated with that set. In Figure 1 we can see that there is a condition name called "All Reports Submitted" which is associated with the values *yes* and *no*. We consider all of this information to make up a condition called "All Reports Submitted".

The top right quadrant of a decision table is made up of condition entries. The condition entries hold the answers to the questions asked in the condition stub. The bottom right quadrant is made up of action entries. The action entries hold the answers to the action questions asked in the action stub. Each column of answers in the table is called a rule [30]. The set of condition values that make up a rule is called a **condition set**. The set of action values that make up a rule is called an **action set**. Each rule indicates which conditions must be met (condition set) in order for the actions indicated in the rule (action set) to be performed [29]. When the events and states in the system (called a transaction), match the conditions specified in a rule, the actions in that rule are executed [30].

One thing to note is that a rule can have an explicit answer to a condition listed in the stub, such as *yes*, *no*, *5*, *after 6:00pm*, etc., or can be left blank for that value. A blank value is referred to as a don't-care-entry. Don't-care-entries imply the entire set of possible values for the corresponding condition and so match any value when tested. This is important when we discuss testing the consistency of the table later on [29].

Note that both condition sets and action sets, although given the name sets, are in fact represented as vectors. The order of the values in a condition set, for example, is important, and will have an entirely different meaning if changed around. For example, suppose we have a condition stub containing two questions: *number of students*, and *professors available to teach*. Now say we have a condition set of 1000, 20. This means that there are 1000 students and 20 professors available to teach. This gives us a student to teacher ratio of 50:1. If we mix up the condition set, however, we now incorrectly see that we have 20 students and 1000 available professors. The values by themselves are useless. We need to know the order in which they appear in order to abstract some meaning from the values. Therefore, our condition sets and action sets are in fact, vectors. Note that to some, the strict definition of a vector is an ordered set of scalars. Montalbano has decided for simplicity to overlook this restriction and use vector as meaning an ordered set of some type of element, be it rules, conditions, actions, etc. All statements in this thesis referring to vectors use Montalbano's wider definition of vectors [29].

There are two types of decision tables. Tables with entry sections expecting to have binary answers to the questions in the stubs such as *yes/no*, *do/don't do*, etc. are called limited-entry decision tables. On the other hand if the answers are expected to be a little more general such as *red*, *5*, *after 6:00pm*, etc., then the table is called an extended-entry decision table. Tables which contain both types of answers are called mixed-entry tables [29]. Note that wild-card or don't-care entries represent two or more values implicitly, therefore tables which contain wild-card entries are considered extended-entry tables, even if all other values are only yes or no [30].

An example decision table giving rules on how to set up a thesis defence is shown below in Figure 1.

	Rule1	**Rule2**	**Rule3**	**Rule4**
All Reports Submitted	YES	NO	YES	NO
Room Booked	YES	NO	YES	
Chairman Appointed	YES	NO	NO	YES
Book a Room		√		√
Announce the Exam	√	√	√	√
Appoint Chair		√	√	
Find Examiners		√		√

Figure 1 - An Example Decision Table

Notice that there are no rules with duplicate condition sets. Theoretically there could be, but there is no need for their separate existence because they could be merged. Suppose we were to have two rules with identical conditions indicated, but different actions specified. Then we could simply merge the two columns such that there is one column with the union of the actions listed in the two old columns. To the contrary, there could be two columns with identical actions but different conditions. This is because it is more difficult to merge columns based on differing conditions, although it is possible.

There are two types of rules in a decision table: simple rules and composite rules. Simple rules have at most one simple value listed in the condition entry. An example of this would be a rule with one condition, *Time of departure* with a value of *5:00pm*. A composite rule (sometimes called complex rule) has one or more entries with two or more values. An example of this would be a rule with the same one condition, *Time of departure*, with a value of *(5:00pm* $\vee$ *9:00am)* [29]. Note that since a don't-care-entry represents two or more values, a rule containing a don't-care-entry is considered composite [31].

3.2. Proposal for Using Decision Tables as an Internal Policy Represenation

There are two things that decision tables make easy: detecting conflicts between rules in the decision table, and checking to see if the table is complete. Since it is important to check our policies for conflicts and may be important to check them for completeness, the format of decision tables to store policies seems appropriate. We will examine both completeness checking and conflict checking in decision tables and discuss how they can be used with policies. In their standard format, decision tables are not sufficient to store policies. Modifications will have to be made in order to store all the policy information. Furthermore, we will discuss the problem of converting policies from the internal decision table representation to that of executable code.

Note that in this section I will use example condition sets of rules to demonstrate certain ideas. The notation I use is the following:

RuleName: {(value1a, value1b, value1c), ¬(value2a), (*)}

Where:

RuleName is the name of the rule.

The curly braces { and } show the end of the rules values.

Values for a particular attribute are grouped in parentheses and separated by commas. Each set of parentheses are also separated by commas.

The value *(*)* denotes the don't-care-value.

The value *¬(value2a)* denotes *NOT (value2a)* i.e., *(*) – (value2a)*.

Since the actions were not important for our present discussion, I have omitted them in this notation. This notation only shows the condition part of the rule.

3.2.1. Checking for Completeness

When writing policies, it is often important to know whether or not a policy set will cover any possible situation the system will ever come across. A policy set which does this is said to be **complete**. Completeness is something that is often difficult to check when a policy is written at a high level, especially when there are many possible conditions to be met. Decision tables prove themselves very useful in this area.

In order to check if a policy set is complete, we must first know how many simple rules are theoretically possible within our table. Recall that a simple rule has exactly one condition value corresponding to each condition in the stub. Therefore the number of possible simple rules in our table is equal to the number of possible values for the first condition multiplied by the number of possible values for the second condition multiplied by the number of possible values for the third condition, etc. [29] For example, let us say that I have policy describing costs of used cars. I have three conditions in my table: *Colour*, *Year*, and *Make*. Let us say that *Colour* can be one of four values: *black*, *green*, *red*, *blue*; *Year* can be one of twenty values: *1981* through *2000*; and *Make* can be one of three values: *Honda*, *Volvo*, *Toyota*. The number of simple rules possible in this table is equal to the number of possible conditions that can occur in the system, which is equal to 4 (number of possible colours) * 20 (number of possible years) * 3 (number of possible makes) = 240.

Once it is known how many simple rules the table can represent, the next step is to find out how many simple rules the table actually represents. If the table only contains simple rules, this is simply a matter of counting how many rules are in the table. However, often times tables will contain composite rules. Composite rules represent multiple simple rules, and so it is necessary to figure out how many simple rules each composite rule represents. In order to do this, we multiply the number of values entered for each condition in the rule [29]. For example, if I have a rule with the condition values

{(red, black), (1995, 1996, 1997, 1998), (*)}

then the number of simple rules this composite rule represents is 2 (number of values for first condition) * 4 (number of values for second condition) * 3 (number of values for third condition) = 24. Note that since the last value was a don't-care value, it in effect represents all possible values for that condition.

This is not completely correct, however. Let us say that we have two rules with condition sets:

{(red, black) (1995, 1996, 1997, 1998), (*)} and

{(*), ¬(1997, 1998), (*)}

According to the above discussion, the first rule represents *2 * 4 * 3* = 24 simple rules, and the second rule represents *4 * 18 * 3* = 216 simple rules. Since our table can represent *4 * 20 * 3* = 240 rules and we are representing *24 + 216* = 240 rules, we might be tempted to say that our policy set is complete. This is not correct, however. Note that the condition set *{(red), (1995), (Honda)}* can be represented by both rules. Note also that the condition set *{(blue), (1997), (Honda)}* is not represented by our policy set. Our policy set contains overlapping conditions, and therefore some of them have been counted twice. Two rules which do not overlap are said to be **disjoint**. In order to count the number of simple rules represented by our policy, all of our rules must be disjoint. We must then create disjoint rules from the rules in our policy set [29].

First, a set of overlap-rules are created. Overlap-rules are rules generated to express the overlap of rules in the table.

Definition 3.2.1.1:

An overlap-rule R_O is created from two rules R_1 and R_2, such that if R_1 has condition set $X:\{x_1, x_2, \ldots, x_n\}$ and R_2 has condition set $Y:\{y_1, y_2, \ldots, y_m\}$, R_O will have condition set $Z:\{z_1, z_2, \ldots, z_p\}$ where $Z = X \cap Y$.

For example, using the two rules above, the overlap-rule would be *{(red, black), (1995, 1996), (*)}*. Using our technique of counting how many simple rules a rule represents, we can see that our overlap rule represents twelve simple rules.

What we need to do is replace one of the two rules that overlap with one or more rules that represent the old rule without the overlap. To do this we suggest the following procedure:

1) Find the rule representing the overlap in the two rules. Call this rule R_O.

    For each slot in R_O:

    2) Call the position of that slot P. If the slot contains only one value, is the null value, or is the don't-care value, do nothing. Otherwise create a rule. The new rule's values are R_O.

    3) The value in slot P of the new rule is then replaced by:
    the intersection of the value in slot P of R_2 and the complement of the value in slot P of R_O.

4) Eliminate the original R_2.

Example:

We have the two rules:

R_1: {(red, black), (1995, 1996, 1997, 1998), (*)} and

R_2: {(*), ¬(1997, 1998), (*)}

First we find the overlap rule R_O, which in this case is:

R_O: {(red, black), (1995, 1996), (*)}

We look at the first slot in R_O which is (red, black). It is more than one value so we create a new rule R_{2a} which takes its values from R_O.

{(red, black), (1995, 1996), (*)}

Now we replace the value in the first slot with the intersection of the first slot from R_2 and the complement of the first slot in R_O.

$$\begin{aligned} R_2^1 \cap \neg R_O^1 &= (*) \cap \neg(red, black) \\ &= (*) \cap (blue, green) \\ &= (blue, green) \end{aligned}$$

Now we have:

R_{2a}: {(blue, green), (1995, 1996), (*)}

Next we examine the second slot of R_O. It is *(1995, 1996)*, which again contains more than one value so we create another rule identical to R_O.

{(red, black), (1995, 1996), (*)}

This time we replace the second slot's value of the new rule with the intersection of the second slot in R_2 and the complement of the second slot in R_O.

$$\begin{aligned} R_2^2 \cap \neg R_O^2 &= \neg(1997, 1998) \cap \neg(1995, 1996) \\ &= \neg(1995, 1996, 1997, 1998) \end{aligned}$$

Now we have:

R_{2b}: {(red, black), ¬(1995, 1996, 1997, 1998), (*)}

Next we look at the third slot of R_O. It is (*), which is the don't-care value so we do nothing. Finally we eliminate the original R_2.

Now as our rules we have:

R_1: {(red, black), (1995, 1996, 1997, 1998), (*)} and

R_{2a}: {(blue, green), (1995, 1996), (*)}

R_{2b}: {(red, black), ¬(1995, 1996, 1997, 1998), (*)}

From our earlier calculations we determined that originally R_1 had a simple rule count of 24, R_2 had a simple rule count of 216, and that there were 12 rules which overlapped. Therefore we were representing 228 rules.

Our new count is:

R_1) $2 * 4 * 3 = 24$

R_{2a}) $2 * 18 * 3 = 108$

R_{2b}) $2 * 16 * 3 = 96$

$Total = 24 + 108 + 96 = 228$

Therefore our new rules do represent the same number of rules as the old one. If we now find the overlap rules we get:

$R_{O1,2a}$: {∅, (1995, 1996), (*)}

$R_{O1,2b}$: {(red, black), ∅, (*)}

$R_{O2a,2b}$: {∅, ¬(1997, 1998), (*)}

Notice that every single rule has a null value somewhere, meaning that none of our new rules overlap with each other. In this example, with only two rules, it was not really necessary to remove the redundancy between the rules, since we could calculate the overlap simply by seeing how many simple rules the overlap rule represented. In a larger system, however, it may be necessary. Let us say we had three rules A, B, and C. A represents 100 rules, B represents 100 rules, and C represents 100 rules. Let's say the entire system can represent 250 rules. We may know that the overlap-rule for $A \cap B$ represents 50 rules, the one for $B \cap C$ represents 50 rules, and the one for $A \cap C$

represents 50 rules. But it is complicated to determine which of these overlap-rules also overlap. For example we may have over-counted the number of simple rules that are represented by the overlap rules if A $\cap$ B $\cap$ C is not empty. It is much easier to transform the policy set into one that does not overlap at all and then count the equivalent simple rules.

It is not always desirable to have a rule set that covers every single possible combination of conditions. In that case, checking for completeness will always return a negative result. If there is a standard completeness checking mechanism in the system that checks every single table for completeness, an else rule with a null action set can be added to cover any rules not covered by the rest of the table, thereby making the table artificially complete.

3.2.2. *Consistency Checking*

Because of the structure of decision tables, it is easy to see particular combinations of conditions and actions when looking at one. [29] This would certainly make it easier for humans to analyze the rules and determine if they are consistent. If a new rule is added to the decision table or an existing rule is modified, first find all conditions that are indicated by the rule in question. Next go along the row for each condition and find all columns that also indicate that condition. (By indicated we mean they have some value other than the don't-care-value.) Each column that does is a rule that could potentially be triggered at the same time as the updated rule. Out of those rules, any one that has an action that conflicts with an action in the original rule can be said to potentially conflict with the original rule.

However, for our purposes, humans will not be used to check for consistency of the rules. What interests us is whether this structure eases consistency checking in some way the computer doing the checking can use. There has been research on how computer programs can best perform consistency checking using computer algorithms, however, it may first be useful to see if this type of conflict checking will solve the problem of detecting policy conflicts.

As mentioned in the previous chapter, policy is concerned with two overall types of inconsistency: modality conflicts and application specific conflicts. In classic decision table theory, the concern is with two other types of conflicts: intercolumn inconsistency and intracolumn inconsistency.

Intercolumn inconsistency occurs when two rules in the decision table have overlapping conditions. What happens is that when the conditions for the two rules are matched at run time, the system does not know which rule it should act upon [29]. This is also referred to as having an ambiguous table. When the two rules with overlapping conditions have the same actions, the ambiguity is said to be apparent. If the rules specify different actions to be performed, the ambiguity is said to be real [32]. In decision table terminology, having a real ambiguity means the table is inconsistent [35].

It has been said that ambiguities must be detected and removed from decision tables [31]. However, King and Johnson have argued that this is an unnecessary restriction to those applications which require only one action for a set of conditions. They argue that some applications may require that two or more actions be performed when one set of conditions is matched, and that this is perfectly acceptable. Rules that permit this behaviour are called multi-action rules [32]. Lew also used these kinds of rules by providing a vector action set as opposed to a single action for each rule. He also allowed all actions to execute in the case that two rules with the same conditions specified different actions [35].

For our purposes, it is perfectly acceptable to have more than one action executed upon a set of conditions being matched. Not only can this be done if we have one rule specifying multiple actions, but also if two separate rules happen to be triggered at the same time. Therefore, detecting the presence or lack of ambiguities in our table will not lead to concluding whether or not our set of policies is consistent, since our notion of consistency is different from that associated with single-action-rule decision tables.

Intracolumn inconsistency refers to a rule having conflict within itself [29]. For example, say one condition asks what year their driver's license was obtained and another asks for the year of birth. Based on driving laws, we know that people under a certain age cannot have driver's licenses. In that case if there is a rule which has a value for year of birth that does not allow a license and there is a non-null value for the year the driver's license issuance, then we have an inconsistency. For decision tables this often seems to be a conflict between two conditions, but it seems logical that this can be extended to cover an action combination that does not make sense according to the "world" the system operates in. For example, if one action says to delete a file, and another action says to modify the file, we would have an inconsistency. Intracolumn inconsistency checking, therefore, seems to map quite well to the notion of application specific conflicts in policy. How this can be done is not discussed in this thesis and is left for future work.

3.2.3. Storing Policies as Decision Tables

Now that we have seen the benefits of representing policy as decision tables, it is important to know how to transform policies represented in another form to the decision table representation.

For the purpose of storing policy rules in decision tables, we will consider an event trigger to be a specialized form of condition. In this way, we will not need to modify our decision tables to be able to specify which events will trigger particular rules. We can simply make a condition indicating that if the event occurs that the condition has been met. This solves the problem of representing both events and conditions in the table which is only meant to represent conditions. For the remainder of this section the word condition in a decision table will refer to our expanded notion of condition which is either a condition from the rule or an event from the rule. For example in Figure 2, *Room Booked* could refer to either a condition or event.

Since all policy rules may at one time fit into one table, the table must have a row for each possible condition statement that appears in the rules. This does not act as a restriction on the number of condition statements the system can represent. Let us say that we are adding a new rule which has a condition statement never before seen in the system. All that needs to be done is to add a row to the table with the new condition statement. Then the new column can be built for the rule in the normal fashion. Since the other rules didn't care about this condition, we need not modify the other rules. They will already have a blank entry for that condition indicating a don't-care-value.

Similarly the table must have a row for each possible action that rules can perform. As with the condition statements, if a new action appears in a rule which the system has not listed, a new row can be built on demand for that action. Notice that in Figure 2, that if Rule 5 were added to the system and the action *Defend* did not appear in the table, that it could be added to the table. The other rules would get a don't-care-entry in that row, which would not affect them in any way.

Since actions and condition statements can be added to the table as needed, there is no need to examine the rules, and construct the rows all at once before entering the rules. The table can be built incrementally, adding rows as needed. For example, if our system can produce 10 events but we are unaware of how many events are possible at the time of building the table, there will be no problem. The table is created using the first rule which let us say indicates two events, so the table now contains two events. Now we take the second rule which let us suppose indicates two different events. These new events are added to the table as the second rule is added to the table and the process is continued, adding new conditions and actions as the rules containing them are processed.

Our table is missing two important attributes that are represented in the PSL we are using, however: the notion of modality, and the order of execution. In Sloman's PSL, each rule has a single modality value. Therefore, we extend the decision table structure such that each column in the entries section has a value representing the modality. The question is where should this value be located. Modality is not something that affects whether or not a rule should be triggered, so it really shouldn't appear in the condition part of a rule. However, it is not really an action either, but a modifier on the actions. It determines whether the actions should be executed, permitted to execute, prohibited to execute, or must not execute. In that case, we should really have another horizontal divider in our table. This third section will hold exactly one row representing the modality values for all the rules. To fit with the rest of the table, the modality section of the table should have a modality stub. Since there is only one row, the item in the stub will never change, and will in fact have no real purpose except to be consistent with the rest of the table. We'll call this stub item ***Modality***. The value in the entries section corresponding to this stub can be exactly one of the values used in Sloman's PSL for each rule: A+, A-, O+, or O-.

If two rules are triggered at the same time, we may wish there to be priority values associated with each rule as suggested by Sloman, in order to determine which rule should be fired first. Therefore, we need another modification to the decision table in order to indicate this priority value. Again, this value is neither a condition upon which we can determine if the rule should fire, nor an action to perform. Therefore we should create another section in our decision table for this purpose. As with the *Modality* row, we will create another row with the stub called ***Priority***. The entries in the *Priority* can be varied depending on the sort of priority used in the policy specification language. It could be numeric values, strings such as HIGH, MEDIUM, LOW, or some other form of priority indicator. The table, being only an intermediate form for the policy to be stored and analyzed, does not need to restrict the values it can accept.

The order of appearance of actions in a decision table rule does not normally affect the order of execution of these actions. If it did, then a particular action X would always be performed before another action Y because actions in the same table always appear in the same order. This may be contrary to what is desired for the behaviour of the rules. Therefore, we modify the action entry of the decision table such that it gives us a way to represent the desired sequence of execution of the actions. The table should allow each action to have a corresponding value representing when the action is to be executed compared to the other actions. There are two ways this can be done: absolute ordering and priority ordering. Using absolute ordering we can specify that a particular action will be the first to execute, another action will be the second to execute, etc. Using relative ordering, we can specify execution priorities for each action. That way, some actions could have the same priority, when it is not important which is executed first. Then, if there are multiple actions of the same priority, there is no need to wait for one if it cannot execute right away. The other actions of equal priority can be executed in the mean time, and waiting will resume if the remaining action still cannot execute.

	Rule1	**Rule2**	**Rule3**	**Rule4**	**Rule5**
Modality	A+	A+	A+	A+	A-
Priority	MEDIUM	MEDIUM	MEDIUM	HIGH	MEDIUM
All Reports Submitted	YES	NO	YES	NO	NO
Room Booked	YES	NO	NO	YES	
Chairman Appointed	YES	NO	YES		
Book a Room		3	2	1	
Announce the Exam	√	1		2	
Appoint Chair		4			
Find Examiners		2	1	3	
Defend					√

Figure 2 - An Example of our Modified Decision Table Format

Figure 2 shows an example of the modified decision table we have just described, with the new and modified areas shaded. Notice that in Rule5 there is a priority value associated with the rule, but no priority value given in the action entry. If there is no action priority value given, the system can determine in which order to perform the actions.

Our next concern is how the policies should be grouped. A policy rule is stored in a decision table as a rule in the table. How to store entire policies or policy sets is not as obvious a choice. A single decision table could be used to store the rules of one single policy, all the policies in a particular policy set, or all the policies in the entire system. There is really no restriction in this respect. However, large decision tables could become unwieldy in terms of memory usage, thereby increasing the time needed to update the table or analyze it for consistency [33]. Therefore, it may be beneficial to store each policy as a separate decision table. This is also advantageous because in order to check for local consistency, often only one policy will need to be compared with itself, so no unnecessary policies will have to be examined if all the rules related to the policy are contained within that decision table. On the other hand, storing entire sets of policies (such as all the policies of a university department) in a single decision table can help to organize policies by keeping related policies together.

In the case where multiple policies are needed in order to check for consistency, two or more tables can be temporarily merged into one larger table for the purpose of analysis. Research has been done on how to best merge decision tables into larger tables in order to optimize performance times, such as the work of Shwayder [33]. In our case, merging the tables would be a similar process, but for a different reason.

Now that our table contains all the necessary attributes needed to represent our policies, we must figure out how to convert them from their policy definition language, into our table.

3.2.4. Conversion of Decision Tables to Code

We now have our information in a table where it can be analyzed for completeness, correctness, and consistency. Once this has been done, the decision tables must be transformed into some form that can be executed. There have been several algorithms devised on how to translate the information given in decision tables into an executable code form [34],[35],[36],[37]. Generally these algorithms translate the contents of the decision table into a flowchart or execution tree.

However, with declarative languages such as CLIPS [38], JESS [39], and PROLOG [40], we could simply form rules in the desired language directly from the rules in the decision table. This would also take care of the problem of deciding exactly how the rules should be executed. Rule engines have been developed over many years with algorithms designed for the execution of declarative rules. In that case, all we need is a translation from the policy terms in our decision table rules to actual executable terms in the declarative language.

Vanthienen and Wets [41] have discussed how easy it is to convert decision tables into rules that are executable by an expert system shell. For example we could use column based translation. This involves creating one rule in the expert system shell for every column or rule that we have in the decision table. This is the most obvious translation method, although they also discuss two other translation methods into expert system rules: entry based translation and row based translation. Entry based translation is where each entry in the table is turned into a rule in the expert system shell. Row based translation creates a rule in the expert system shell for every action indicated in the decision table. Therefore the problem of converting decision tables to executable rules has already been solved, and we can benefit from these research results.

3.3. Summary

The power of decision tables is the ability to check its rules for completeness and to check for consistency. Since rules from decision tables can easily be adapted from policy rules, the decision table format seems like a natural method of storing policies in a declarative manner. Although the consistency checking normally performed on decision tables does not map perfectly to the problem of consistency checking between policies, they do serve as a helpful tool in performing completeness checking. With rule engines available, all that is required to make policies executable is to take the decision table rules and translate them to the expert system shell language.

4. Incremental Validation

4.1. *Why is Incremental Validation Needed*

Clearly stated, the problem of incremental policy validation is this: Assume that there is a pool of rules R. Let us say for now there are a thousand of them. Assume that these rules have been validated somehow to check for conflicts, and R has been determined to be consistent – i.e., there are no conflicts within R. Suppose a change, ΔR, is made to R, such that the number of rules that have changes are much smaller than the total number of rules in R. We want to see if the set of rules R is still consistent without having to revalidate the entire set. We are aiming to revalidate R with significantly less effort than is needed to revalidate the entire set of rules. In particular we will show that this can be done in linear time.

4.2. *Simple Incremental Conflict Detection*

Let us assume that our system does not have all the properties of a distributed system, in particular the property that events multiple events can occur at the same time. We assume that events cannot happen concurrently. One event must occur before or after another, even if they are separated by only an extremely small time interval.

Then, two rules can be triggered at the same time if and only if they contain the same triggering event in their conditional statements. Furthermore, the common event between them must be the final event to occur out of all the events listed in the conditional statements for both rules. This is because we assume that two events cannot occur simultaneously. For example: if R_1 fires upon events A, B, and C occurring, and R_2 fires upon events A, D, and E occurring, then A must be the event that occurs last if these two rules are to conflict. If events A, B, and D are fired, then if event C or E occurs, then one of the rules will be triggered. Since both events cannot occur at the same time, only one of the rules will be triggered at a time. The only way the two rules can be triggered at the same time is if events B, C, D, and E were fired, and then event A is fired last. Note that in the condition section of our decision table we are including both the event triggers and the conditions obtained from the policy specification language. Therefore for the remainder of this chapter, when we refer to a condition in the decision table, we are referring to either a policy's condition or a policy's event. There will be no difference between the two in our conflict detection algorithm.

Theorem 1:

Two rules can be triggered at the same time if and only if the condition entries for both rules contain at least one common condition.

Corollary 1:

Out of all the conditions in the condition entries for both rules, one of the common conditions must occur last in order for both rules to be triggered at the same time.

Theorem 2:

If a set of rules S is the union of two sets of rules S_1 and S_2, where S_1 is consistent, and S_2 is inconsistent, then S will be inconsistent.

Proof:

If S_2 is inconsistent, that means that S_2 contains two or more rules such that they conflict with each other. Since S is the union of S_1 and S_2, S will also contain this set of conflicting rules. Therefore S is inconsistent.

Corollary 2:

Any set of rules containing a rule which is inconsistent with itself, is an inconsistent set.

Example: if rule R_1 states that it should lock door A and open door A, this is inconsistent. Therefore the entire set of rules S in which R_1 is contained is inconsistent.

Theorem 3:

If the set of rules S is the union of the sets of rules S_1 and S_2, where S_1 is consistent, and S_2 is also consistent, then there is no conclusion on whether S is consistent or not. It could be either consistent or inconsistent.

Proof:

Say S_1 contains one rule R_1: on event X open window A.

Say S_2 contains one rule R_2: on event X close window A.

It is quite obvious that S_1 is consistent, since there are no other rules in S_1 and R_1 does not conflict with itself. The same can be said of S_2. Now if we take $S = S_1 \cup S_2$, we can see that R_1 and R_2 will be triggered at the same time, and that they have conflicting actions. This means that if event X occurs, that a conflict will occur between R_1 and R_2. Therefore S is non-consistent. Thus, the union of a consistent set of rules with another consistent set of rules does not necessarily yield a consistent set of rules.

Theorem 2 and Theorem 3 lead us to two important observations. From Theorem 2 we can see that if we find a rule which is inconsistent with itself, then we can stop the validation process and declare that the set of rules is inconsistent. From Theorem 3 we can determine that if we add a new rule to a set of rules previously said to be consistent, we cannot say that the new set is consistent without re-validating the set, even if the rule is consistent with itself.

Therefore our incremental validation algorithm can begin with the following step:

See if a rule is consistent with itself. If it is not, then we can determine that the system is inconsistent. If the rule is consistent with itself, then move onto the next validation step.

There are two ways a rule can be inconsistent with itself: if it contains mutually exclusive conditions, or if it contains opposing actions.

Mutually Exclusive Conditions

If a rule depends on two conditions that can never occur together, then the rule is said to be inconsistent. An example of this would be if a rule had a condition "`(x = '6') ^ (x = '9')`". These two statements can of course never be true at the same time, so we know that the rule will never be triggered. Although this will not result in a runtime conflict, it means that the rule is useless as it will never be fired, and so the user should be warned. Therefore we consider this rule inconsistent.

Opposing Actions

If a rule contains two actions that oppose each other, then the rule is said to be inconsistent. An example of this would be if a rule had an action that said "Close Door A" and another action that said "Open Door A". The end result of these actions is completely dependent on the order that these two actions are performed. Therefore we consider this rule to be inconsistent.

The next step in our algorithm would be to check the new rule to make sure it is not inconsistent with any other rules.

Algorithm V_0:

1. See if the new rule R_N contains two conditions that can never occur together. If so, halt, and return that the new rule is inconsistent.
2. See if the new rule R_N contains two actions which oppose each other. If so, halt, and return that the new rule is inconsistent.
3. For each rule in the policy system, compare it with R_N to see if they have a common condition and an opposing action. If so, halt, and return that the two rules conflict.

Given that there are *n* rules in the system, this process would take O(n), since the new rule needs to be compared with every rule in the system. Compare this with complete revalidation which compares every rule with every other rule, taking O(n^2). Algorithm V_0 is effective, but it compares the new rule with rules which have no chance of conflicting. There must be a more effective way to validate our system of rules.

4.3. A Simple Improvement to Validation

In the algorithm V_0, every single rule in the system was compared with the rule that was changed. We could improve our algorithm by comparing the new rule only with rules that have a possibility of conflicting. We must find a way to determine which rules should be examined.

We know that in order for two rules to conflict with each other, that they must be triggered at the same time. From Theorem 1 we know that in order for two rules to be triggered at the same time, they must both contain at least one common condition. Therefore if two rules do not have any conditions in common, we know they can never conflict, and thus we do not need to compare these two rules when validating.

This makes the set of rules that we need to compare smaller, but can we shrink the set even further. In order for two rules to potentially conflict, two conditions must be met. As mentioned above, the first condition is that both rules contain at least one common condition to trigger them. This means the rules can both be triggered at the same time. The second condition is that each rule must contain an opposing action with the same modality, or the same action with opposing modality. This means if the rules are triggered at the same time, each rule will attempt to do the opposite of the other rule. Depending on which rule we choose to perform first, the outcome will be completely different. Therefore we can compare the new rule to two subsets of rules:

(a) those rules which have an action opposite to one in the new rule but the same modality.

(b) those rules which have an identical action to one in the new rule but has the opposite modality.

Any rule which is in both of these subsets does not even need to be compared with the new rule, since it automatically meets the definition of potentially conflicting with the new rule. Therefore, it is enough to know which rules fall in both of these subsets of S. Then we can find all the rules that potentially conflict with the new rule.

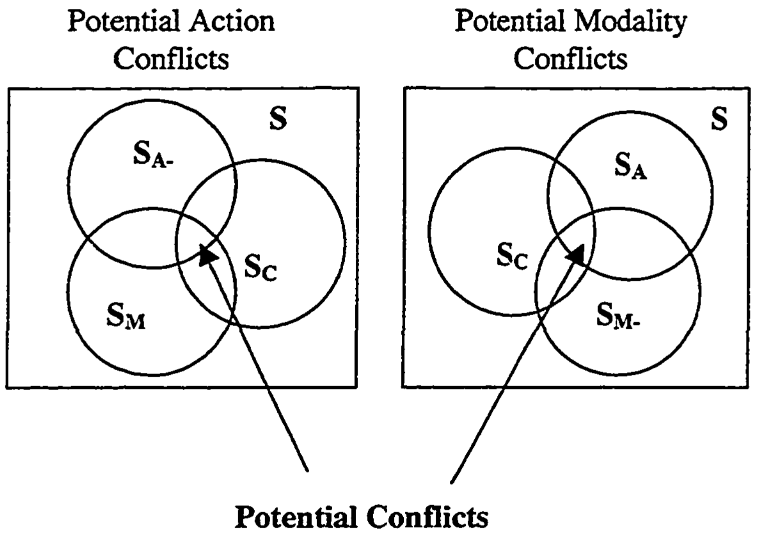

Figure 3 – Potentially Conflicting Rules with R_N

The following explains the symbols in Figure 3:

S : The entire set of rules in the system

S_C: The set of rules that contain a common condition with the new rule, R_N

S_A: The set of rules that contain an action found in R_N

S_{A-}: The set of rules that contain an action opposite to R_N

S_M: The set of rules with the same modality value as R_N

S_{M-}: The set of rules with an opposing modality value to R_N

$C_A = S_C \cap S_M \cap S_{A-}$: The set of rules with a potential action conflict with R_N

$C_M = S_C \cap S_{M-} \cap S_A$: The set of rules with a potential modality conflict with R_N

4.3.1. Development of the Concepts for Algorithm V_1

Let us examine a policy set in decision table form, and see how we can determine which rules potentially conflict. Let us say that initially the table was validated as consistent, and we added a new rule, R_1. We must now find out if R_1 potentially conflicts with any rules in the table. Let us begin by constructing an exact copy of our decision table, which we can manipulate to eliminate rules we know don't match conflict criteria. (See Figure 4.)

	R_N	R2	R3	R4	R5	R6
M	A+	A-	A+	A-	A+	A-
C1	√			√		
C2			√	√		
C3	√		√			√
C4		√		√	√	
C5					√	√
A1			√			√
A2	√	√	√			
A3	√			√		
A4				√	√	√

Figure 4 – Decision Table Before Validation Process

	R_N	R2	R3	R4	R5	R6
M	A+	A-	A+	A-	A+	A-
C1	√			√		
C2			√	√		
C3	√		√			√
C4		√		√	√	
C5					√	√
A1			√			√
A2	√	√	√			
A3	√			√		
A4				√	√	√

Figure 5 – All Irrelevant Rows are Eliminated

The first thing we know is that in order for a rule to conflict with R_1 it must contain at least one common condition with R_1. Therefore we can eliminate all conditions from the table that R_1 does not have. (See Figure 5.)

Next, we can eliminate the rules whose condition entries are all empty. (See Figure 6.)

	R_N	R2	R3	R4	R5	R6
M	A+	A-	A+	A-	A+	A-
C1	√			√		
C3	√		√			√
A1			√			√
A2	√	√	√			
A3	√			√		
A4				√	√	√

Figure 6 – All Empty Condition Entry Rows are Eliminated

	R_N	R3
M	A+	A+
C1	√	
C3	√	√
A1		√
A2	√	√
A3	√	
A4		

	R_N	R4	R6
M	A+	A-	A-
C1	√	√	
C3	√		√
A1			√
A2	√		
A3	√	√	
A4		√	√

Figure 7 – Table is Split by Modality

The next thing we know is that in order for two rules to have a modality conflict they must have opposite modality values. In order for two rules to have an action conflict they must have the same modality values. Therefore, we must examine all rules that have the opposite and the same modality values to R_N, and can ignore the rest. Let us create a new table containing all rules with the opposite modality value as R_N. Let us create another table containing all rules with the same modality value as R_N. (See Figure 7.) We are creating two tables instead of one because the actions that follow on each of the tables will not be the same. Note that if our new rule was of modality O+, then we would split our table into three tables: one for O+, one for O-, and one for A-, since we can have O+/O- and O+/A- modality conflicts.

In the table that contains rules with the opposite modality value as R_N, we are looking for modality conflicts. In order for two rules to have a modality conflict, they must contain at least one common action. Therefore let us eliminate all actions in the table that are not found in R_N. Similarly in the other table, the one with the same modality value as R_N, we are looking for action conflicts. In order for two rules to have an action conflict, one rule must contain an action that is an opposing action to an action in the second rule. Therefore let us eliminate all actions in this table that are not opposing actions to an action found in R_N. (See Figure 8.)

	R_N	R3
M	A+	A+
C1	√	
C3	√	√
A1		√
A2	√	√
A3	√	
A4		

	R_N	R4	R6
M	A+	A-	A-
C1	√	√	
C3	√		√
A1			√
A2	√		
A3	√	√	
A4		√	√

Figure 8 – All Non-Opposing Actions are Eliminated

Once this is done, we can again eliminate some rules from each of the tables. This time we remove all rules whose action entries are all empty. (See Figure 9.)

	R_N	R3
M	A+	A+
C1	√	
C3	√	√

	R_N	R4	R6
M	A+	A-	[illegible]
C1	√	√	
C3	√		√
A2	√		
A3	√	√	

Figure 9 – All Empty Action Entry Rules are Eliminated

Finally, we remove R_N from the table with the same modality value as R_N, and the other table if it still exists. Now any rules that are left in either of the two tables are rules that potentially conflict with R_N. In Figure 10 we can see that R_4 has potential modality conflict with R_N.

	R_N	R4
M	A+	A-
C1	√	√
C3	√	
A2	√	
A3	√	√

Figure 10 – Only the Conflicting Rules Remain

Let us examine what we know about the table with the same modality value as R_N. We will call this table T_A. We know that any rule found in T_A has at least one condition in common with R_N. In fact, we know that the conditions it has in common are the conditions that are indicated in T_A after performing the steps above. We also know that any rule in T_A has at least one action which is an opposing action to one found in R_N. Again we also know exactly which actions they are: they are the ones indicated in T_A. Finally we know, of course, that any rule in the table has the same modality value as R_N. These rules then meet all the requirements necessary to have an action conflict with R_N, therefore we know that all these rules have a potential action conflict with R_N. All the actions and conditions that would be responsible for each rule's conflict are the non-empty entry values in each rule in T_A.

Similarly, in the table with the opposite modality value as R_N, which we will call T_M, we know that any rule found in this table has at least one condition in common with R_N. As with T_A, the common conditions are indicated in the table. We also know that any rule in T_M has at least one action in common with R_N, and again, these actions are indicated for each rule in the table. Finally, of course, we know that each rule in T_M has the modality value opposite to R_N. These rules meet all the necessary requirements to have a modality conflict with R_N, therefore we can say that these rules have a potential modality conflict with R_N. Again, all the actions and conditions responsible for each rule's conflict are the non-empty entry values in each rule in T_M.

4.3.2. Algorithm V_1

R_N: the new rule

D: the input decision table

Algorithm $V_1(R_N, D)$:

1. Create a duplicate of the policy decision table D. Call this table D'.
2. Eliminate from D' any conditions not found in the rule being examined, R_N.
3. Eliminate from D' any rules whose condition entries are all empty.
4. Split D'. Make a new table T_M, containing all rules with the opposite modality to R_N, and a new table T_A, containing all rules with the same modality as R_N. Eliminate all other rules from D'.
5. In T_A, eliminate any actions that are not opposing actions in R_N. In T_M, eliminate any actions that are not found in R_N.

6. In T_A and T_M, eliminate any rules whose action entries are all empty.
7. Eliminate R_N from T_A and T_M.
8. The rules that remain potentially conflict with R_N, and the non-empty entries correspond to the conditions and actions that overlap with R_N. Table T_A contains rules that have action conflicts with R_N. Table T_M contains rules that have modality conflicts with R_N.

4.3.3. Analysis of V_1

We should see if we are doing each step in the right order to obtain the most efficient solution. First we will examine whether the table should be divided before or after removing the conditions. If we divided the table into two smaller tables before we had removed any non-relevant actions, we would have to remove the same conditions for each table. The way we do it, we are only removing each condition once. Therefore, by removing the conditions before dividing the table, we are being twice as efficient as compared to removing them after dividing.

We next examine whether the table should be divided before or after removing the actions. If we divided the table into two smaller tables after we had removed any non-relevant actions, we would have had a lot more work. Since in each table, different kinds of rules were removed, we would not have been able to remove rules and then divide the table. When we pick which rules to remove, we are deciding what kind of conflicts we are looking for. By removing, say, all rules that don't have identical actions to those in R_N, we are choosing to look for modality conflicts. Therefore, once this step is done, the table can no longer be used to find action conflicts. In order to find action conflicts, we would have to start the entire process from D again. Therefore, by dividing before removing actions, we save doing the entire procedure twice. By similar arguments, it is easy to see that removing irrelevant conditions before removing actions is also the more efficient approach.

Say we had a table with C conditions, A actions, and n rules. Our approach starts by examining each condition once and eliminating it if necessary. This takes C operations. Next each rule is examined to see if its condition entries are all empty, and the rule is removed if so. This takes n operations. Let us assume as the worst case that no rules were removed in the previous step leaving us with n rules. The table is then split into two smaller tables, by examining each rule and putting all rules of opposite modality into one table and all rules of the same modality into another table. Let us assume the worst case, that all rules in the original table had a modality value either the same as or opposite to R_N's modality value. Then this takes n operations and all the rules are still present. Say that the first new table contains n_1 rules and the second contains n_2 rules. Next each action is examined to see if it is a possible conflicting action. This is done for each action in each of the two tables so this takes $2 * A$ operations. Finally each rule is examined in each of the tables, to check if all its action entries are empty, and the rule is removed if so. This takes n_1 operations for the first table and n_2 operations for the second table. Note that $n_1 + n_2 = n$, therefore this step takes n operations. Then we can say that our entire process takes $C + n + n + 2 * A + n = C + 2A + 3n$ operations in the worst case.

Let us say we had done the validation by comparing R_N with each other rule in the system (Algorithm V_0). For each rule R_I compared with R_N, we must examine each condition of R_I which takes c_I comparisons, and each action of R_I which takes a_I comparisons. Assume c_I has an average value of c_{avg} and a_I has an average value of a_{avg}. Therefore each rule compared with R_N takes $a_{avg} + c_{avg}$ operations. Assuming we have n rules in the system, there are $n - 1$ rules to compare with R_N, therefore $(a_{avg} + c_{avg}) * (n - 1)$ operations to perform. This method takes $a_{avg}n + c_{avg}n - a_{avg} - c_{avg}$ operations, while our algorithm using decision tables takes $C + 2A + 3n$ operations. Keep in mind that the number of conditions and actions that are possible in a system generally stay constant, therefore as more rules are added to the system, Algorithm V_1 takes only 2 more operations per rule. Compare this to Algorithm V_0. In the absolute best case each rule has one single condition and one single action, therefore the $a_{avg}n + c_{avg}n$ becomes $2n$, and so as n increases, the number of operations needed is less than Algorithm V_1. If however, there is more than 1 action or condition per rule, then the number of operations rise at a rate greater than or equal to Algorithm V_1. Therefore except under the rare condition where each rule only has one condition and one action, Algorithm V_1 will perform better for large *n* values.

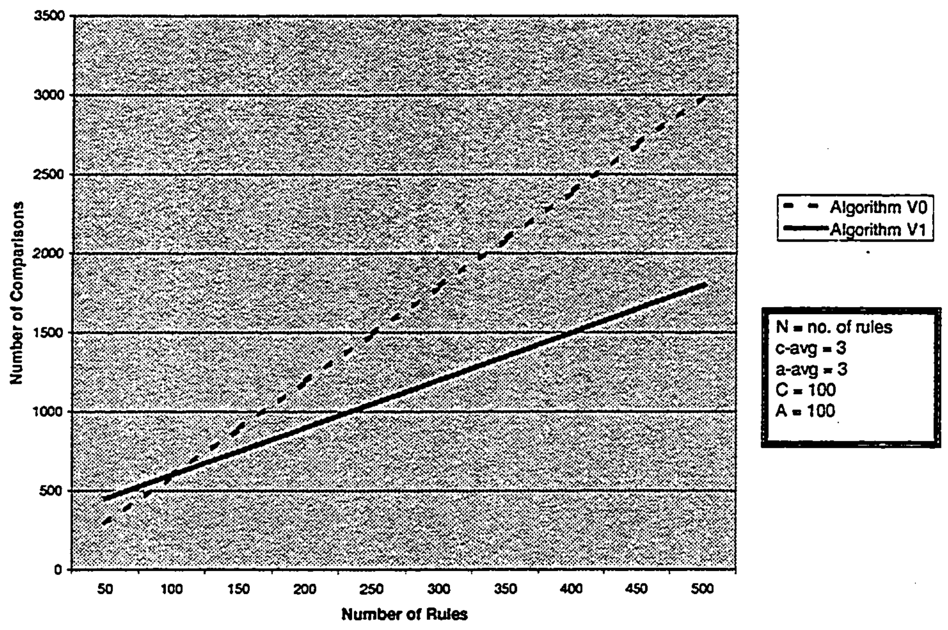

Figure 11 – Graphical Comparison of V_0 and V_1

4.4. *Trigger Chaining*

Suppose we know not only which conditions and actions a rule contains, but also which rules a particular rule may trigger. For example if the execution of R_1 results in an event[1] that is in the condition of R_2, then we can say that R_1 potentially triggers R_2.

Definition 4.4.1:

A rule R_i triggers R_j if and only if there exists an event E, such that E is caused to occur upon the execution of R_i, and E causes R_j to fire.

1 "The execution of R1 results in an event" is abbreviated as "R1 throws an event" throughout the rest of the thesis

Definition 4.4.2:

If there is a rule R_1 which upon firing creates an event E_1, and another rule R_2 which has E_1 as a condition, then we say that R_1 potentially triggers R_2.

Therefore even though a new rule R_N may not contain a condition that is in R_2, if it contains a condition in R_1, then there is a chance that it will conflict with R_2 as well. This conflict does not meet the strict definition of conflict we mentioned earlier. In this case we consider it a conflict because the action in R_2 could undo an action performed in R_N, therefore the action in R_N has no lasting effect. A method to detect these conflicts is needed in addition to conflicts that meet our earlier definition.

Definition 4.4.3:

If there is a rule R_1 which throws *k* events E_1, E_2, ... E_k, and another rule R_2's conditions consist of a subset of the events E_1, E_2, ... E_k, then we say that R_1 triggers R_2. Since R_2's conditions are a subset of all the events thrown by R_1, we know that all R_2's conditions will be met when R_1 fires. Therefore R_1's firing will certainly cause R_2 to be triggered.

We can visualize this by drawing a graph with each vertex representing a rule in the system. Each edge from one vertex to another vertex represents a rule potentially triggering another rule. In Figure 12 we see that R_1 potentially triggers R_2 and R_3, R_2 potentially triggers R_4 and R_5, etc.

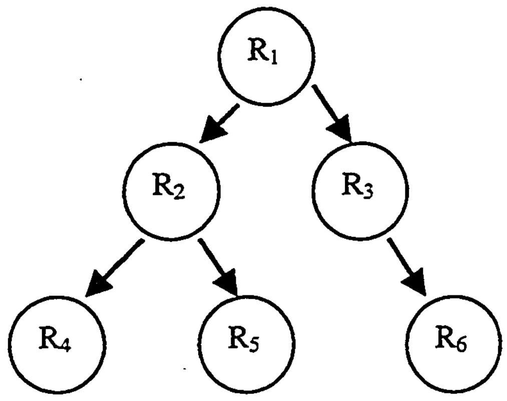

Figure 12 – Trigger Graph for a Set of Rules

Definition 4.4.4:

Suppose a graph is drawn where each vertex represents a rule in a system, and an edge from one vertex to another represents a rule potentially triggering another rule. If there is a path from one vertex V_1 to another vertex V_2 of length *n*, then we can say that R_1 potentially triggers R_2 by degree *n*.

We can now say that if R_1 potentially triggers R_2 by some degree n, and R_2 has an opposing action to an action in R_1, that R_1 and R_2 potentially conflict.

4.4.1. Algorithm V_2

To detect the above kind of conflict, first we start by adding R_N to our graph of the system. We do this by adding a node to represent R_N. Then we draw an arrow from R_N to any rule that contains an event in its condition that is found in R_N's action entry. This is, however, only half done. This connects R_N to the nodes that it potentially triggers, but it does not include the rules that potentially trigger R_N. Suppose we had R_A which had opposite modality to R_N and actions in common with R_N but did not have any conditions in common. Then looking at these two rules alone, they will not be triggered at the same time, and therefore do not potentially conflict. Let us say, though, that there is a rule R_B which causes events 1 and 2 to occur. Lets us also say that R_N gets triggered upon event 1 occurring, and R_A gets triggered upon event 2 occurring. Therefore, when R_B fires, R_A and R_N potentially will be triggered and could have a conflict. Then it is not only important to look at the tree from R_N down, but to examine the whole tree that R_N is in. So we must make the tree complete.

One way to do this is to keep a list of all rules that cause each event to occur, sorted by event. Then when we add R_N to the graph, we see what events cause it to fire. In this case Event 1. Then we look up Event 1 in our list and see that rules R_B and R_Z cause Event 1 to occur. Then we draw arrows from R_B to R_N and from R_Z to R_N. Then the graph is complete. We must keep track of the roots of the trees we have just connected R_N to. These trees will be used to find any conflicts.

Let us first examine the situation to see which nodes in our graph must be validated. Suppose we have the graph in Figure 13.

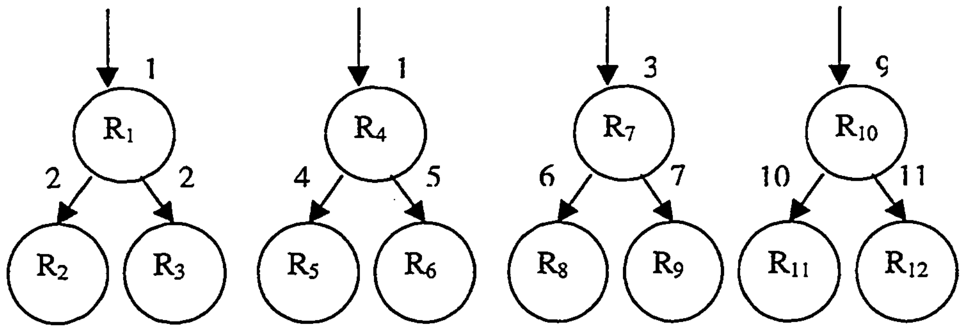

Figure 13 – A Consistent Trigger Graph

In Figure 13 we see that rule R_1 is triggered upon receiving event 1, and upon being triggered, causes event 2 to occur. This triggers R_2 and R_3 to occur, and so on. Suppose that we have previously validated these rules and so no rule in our graph potentially conflicts with any other rule in our graph. This is a consistent set of rules.

Now let us add a new rule, R_N, to the system, resulting in the system shown in Figure 14.

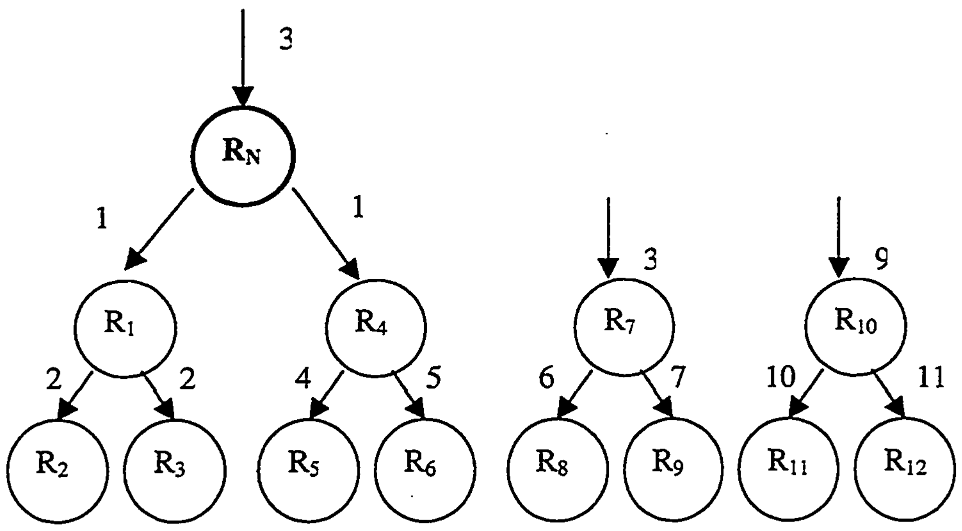

Figure 14 – New Rule Added Above Two Trees with Common Events

As we can see in Figure 14, event 3 causes R_N to fire, which in turn triggers R_1 and R_4. All the rules in the tree whose root is R_N are triggered when R_N is triggered, in this case: R_N, R_1, R_4, R_2, R_3, R_5, and R_6. This means there is the possibility of R_N conflicting with any of these rules. Therefore we must compare R_N with every rule in the tree containing R_N to see if there is a conflict. Not only this, but we see that event 3 causes not only R_N to be triggered, but also R_7. Since none of the rules in R_N's tree were previously triggered upon event 3 occurring, all these rules must be checked to see if they conflict with any of the rules in R_7's tree. In other words, one of the rules in the set {R_N, R_1, R_4, R_2, R_3, R_5, R_6} could conflict with one of the rules in the set {R_7, R_8, R_9}. Since R_1 and R_4 are triggered by the same event, they would have been triggered at the same time, before we added R_N. We already knew that the system of rules was consistent before R_N was added, with R_1 and R_4 being triggered by the same event, there is no change, now that they are both triggered if event 3 occurs as well. Therefore no conflicts will arise between rules in R_1's subtree and R_4's subtree.

Now let us assume we add R_N such that we get the situation depicted in Figure 15.

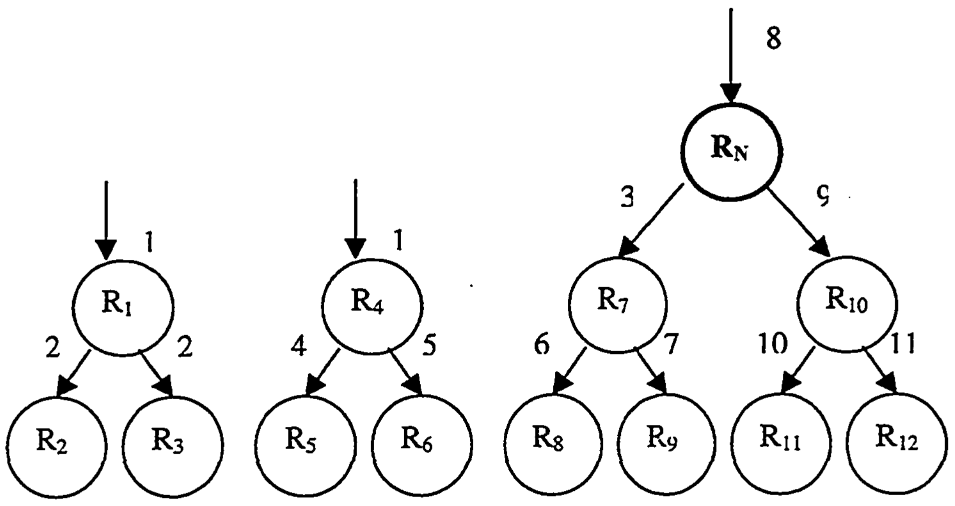

Figure 15 – New Rule Added Above Two Trees with Different Events

In Figure 15 we see that event 8 triggers R_N. R_N causes events 3 and 9 to occur, thereby triggering rules R_7 and R_{10} respectively and so on. As in the previous case, the rules in R_7's subtree and the rules in R_{10}'s subtree must be checked to see if they conflict with the new rule, R_N. This time, however, the two subtrees are not triggered by the same event as in the last case. Although the system was consistent before R_N was added, that conclusion was based on R_7 and R_{10} being triggered by different events, and therefore they could never be fired at the same time. Now R_N firing could cause both R_7 and R_{10} to fire at the same time. This means that the rules in R_7's subtree could possibly conflict with the rules in R_{10}'s subtree. This must be examined. In summary, for this case, it is important to check if R_N conflicts with any of the rules in its tree. Also we must make sure that the rules in every child tree of R_N are consistent with the rules in every other child tree of R_N that is triggered by a different event.

Now suppose we add R_N such that we get the graph shown in Figure 16.

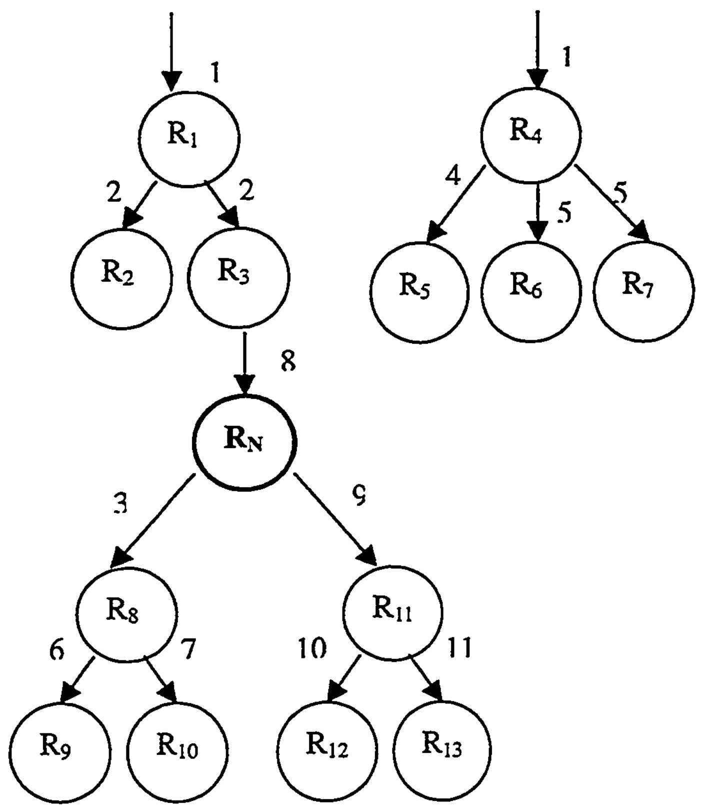

Figure 16 – New Rule Added Above Two Trees and Below Another

In Figure 16 we have added R_N such that it is triggered by event 8, thrown by R_3, and throws events 3 and 9 itself, thereby triggering R_7 and R_{10} respectively. This has added R_N to a tree, where R_N is not the root. As always we must check to see if R_N will conflict with any of the rules in its tree. This time, since R_N is not the root, note that it is not just R_N children we must check against R_N. In this case when the rules in the original tree that R_N was added to, the rules in R_N's subtree are also triggered, which was not the case before. Therefore we must check the rules in the original tree to see if they conflict with R_N's subtree.

Since R_N triggers two rules with different events, as in the last case, we must check that each child tree does not conflict with every other child tree of R_N.

Notice that the root of the tree R_N is contained in R_1. This means that when R_1 is triggered, so are all the other rules of the tree. Therefore, since there are new additions to this tree, we must see if this tree conflicts with any other tree triggered by the same events. In this case R_4 is triggered by event 1 which triggers R_1. Therefore there could be conflicts between rules in R_1's tree and R_4's tree. Notice, though, that in R_1's tree, all the rules not found in R_N's subtree were originally in that tree. In this case those rules are R_1, R_2, and R_3. Since all these rules were triggered by R_1 in the original system, and the original system was said to be consistent, we know that none of these rules will conflict with the rules in R_4's tree. Therefore the only rules that need to be compared against R_4's tree are those found in R_N's subtree. In summary then, for this case, R_N must be checked to see if it conflicts with any of the rules in its tree. Also any child trees of R_N must be checked to see if they conflict with any child trees of R_N that are triggered by a different event. The tree that contains R_N has events that trigger the root. R_N's subtree must be compared against every tree that is triggered by one or more of these events.

There are three things we can conclude from all this. First of all, we know that no matter where R_N is added, R_N must be checked to see if it potentially conflicts with any of the rules in its tree. Second, if the tree containing R_N is triggered by events $X:\{X_1, \ldots X_n\}$, then all the rules in the subtree with root R_N must be compared with all the rules in any tree triggered by one or more events $X_k \in X$. Third, if multiple trees are merged into one large tree, then the rules of each merged tree must be compared with the rules of each other merged tree. With this information we can build an algorithm to check for the consistency of the rules with the minimum number of comparisons.

In order to find all conflicts, we must perform three steps:

1. Find any conflict between a rule in the original tree R_N was added to, and R_N's new subtree.
2. Find any conflict between rules in R_N's subtree and rules in a tree whose root is triggered by an event that also triggers the root of R_N's tree.
3. Find any conflicts that occur between children trees of R_N.

4.4.2. Step 1

Let us start by creating a decision table that contains all the rules in R_N's subtree, including R_N itself. Let's call this table DT_N. We want to compare these rules with the rules in the original tree that R_N was added to. We can do this by adding one rule from the original tree to DT_N at a time, and using Algorithm V_1 to determine if there are any conflicts after each addition. This means, though, that Algorithm V_1 will need to be executed once for every rule in the original tree. There may be many rules in the original tree and this could end up being a time consuming process. We should try to improve on this approach. Consider the following situation:

4.4.2.1. Improvement to Step 1

1. Make an empty rule for each modality that can exist in the system. In our case we have a rule for A+, one for A-, one for O+, and one for O-.
2. Next traverse every rule in R_N's subtree. When a rule is traversed, merge all of its actions with the rule along with the corresponding modality. At the end of this process, we will have four rules, each containing all the actions found in R_N's subtree for each modality in the system. We will call the rules R_{NA+}, R_{NA-}, R_{NO+}, and R_{NO-}.
3. Repeat steps 1 and 2 for the original tree to which R_N was added. This time we get the rules R_{OA+}, R_{OA-}, R_{OO+}, and R_{OO-}.
4. Now we can compare pairs of these new compilation rules, one from each tree, to check for conflicts. To check for modality conflicts the following pairs of rules should be examined to see if they contain any common actions:

   R_{NA-} and R_{OA+}
   R_{NA+} and R_{OA-}
   R_{NO-} and R_{OO+}
   R_{NO+} and R_{OO-}
   R_{NO+} and R_{OA-}
   R_{NA-} and R_{OO+}

To check for action conflicts the following pairs of rules should be examined to see if they contain any opposing actions:

R_{NA-} and R_{OA-}

R_{NA+} and R_{OA+}

R_{NO-} and R_{OO-}

R_{NO+} and R_{OO+}

5. If a rule R_{NX} is found to conflict with the rule it was compared to we know that there is a conflict between one of the rules in R_O's tree and one of the rules in R_N's tree, and we continue on to the next step. Otherwise, we know there are no conflicts between those two sets of rules and we can stop.
6. Take R_{NX} and use Algorithm 1 using the rules in R_O's tree, to see which rule in R_O's tree conflicts with it. Call this rule R_{OC}.
7. Once R_{OC} has been found, use Algorithm 1 using the rules in R_N's subtree to find out which rule R_{OC} conflicts with. Call this rule R_{NC}. We now know which two rules conflict with each other in R_O's tree and R_N's tree.

If two rules are detected as potentially conflicting, then we can stop our conflict detection, and report the two rules that conflict. Otherwise, we must continue to Step 2.

4.4.2.2. Explanation of Improvement

Let us examine how Step 1 works. By creating the four rules, we are compiling all the actions that will occur based on R_N being triggered and matching them with the appropriate modality. Since we want to examine the worst case, when all rules in the tree will be triggered, we can ignore the conditions of the rules. In this way we can think of R_N's tree as being all one rule, with all the possible modality values. We are simply dividing this virtual rule into four real rules, each having one single modality value. In order for a conflict to occur between R_N's tree and a tree that gets triggered at the same time as R_N, there must be two actions in each of the trees that conflict. That is to say that either both trees contain a similar action with opposing modalities, or that both trees contain an opposing action with identical modalities.

We then compile the four summary rules for the original tree that R_N was added to. Call the rule at the root of this tree R_O. Now we have four rules for each tree. R_{OO+} contains all actions that will be performed for modality O+ when R_O is triggered (prior to R_N being added to the system). Similarly we have R_{OO-}, R_{OA+}, and R_{OA-}, for actions that will be performed when R_O is triggered for modality O-, A+, and A- respectively.

We now have 8 rules: R_{NO+}, R_{NO-}, R_{NA+}, R_{NA-}, R_{OO+}, R_{OO-}, R_{OA+}, and R_{OA-}. We can now compare R_{NO+} and R_{OO+} to see if there are any actions in R_{NO+} that are opposite to an action in R_{OO+}. If so, we know we have an action conflict between the two trees, since both rules would be triggered by the same events, and are of the same modality. Similarly we can compare R_{NO+} with R_{OO-} to see if there are any identical actions in both rules. If so, we know that there is a modality conflict between the two trees, because we know the two rules are triggered by the same events and are of opposite modality. This can be performed for the other six rules as well. We compare R_{NA+} with R_{OA+}, R_{NO-} with R_{OO-}, and R_{NA-} with R_{OA-} looking for opposite actions, which result in potential action conflicts. We compare R_{NA+} with R_{OA-}, R_{NO-} with R_{OO+}, R_{NA-} with R_{OA+}, R_{NO+} with R_{OA-}, and R_{NA-} with R_{OO+} looking for identical actions, which result in potential modality conflicts.

By performing these ten comparisons, we can determine if there are any potential action or modality conflicts between the two trees. It does not tell us, however, which two rules are the ones that conflict with each other. Therefore the previous actions are performed in order to see whether or not further conflict analysis needs to be done. It is a simple test to determine if there are potential conflicts present. Provided we have found the presence of potential conflicts, it would be necessary to examine the rules further. We should examine whether it is necessary to compare every rule in R_N's subtree with every rule in R_O's original tree. In fact, it is not necessary.

We know which of the summary rules conflicted, and we can use that as a lead. Say that R_{NO+} conflicted with R_{OO-}. Then we know that there is a rule in R_N's tree with modality O+ that conflicts with a rule in R_O's tree with modality O-. We can take R_{NO+} and compare it with each rule in R_O's tree, until we find a rule that conflicts with it. Let us call this rule R_{OC}. This is one of the two conflicting rules that we are looking for in our system. Then we need to find the corresponding conflicting rule in R_N. We can compare R_{OC} with every rule in R_N's tree until we find a rule that conflicts with it. Let us call that rule R_{NC}.

4.4.2.3. Analysis of Improvement

Let us assume that R_N's subtree contained x rules and R_O contained y rules. If we were to perform comparisons with each rule in R_O with every rule in R_N's subtree, in the worst case there would be no conflict detected, and every rule in one tree would have to be compared with every rule in the other tree. This would take $x * y$ comparisons. Assuming x and y are proportional to n, the total number of rules in the system, this process is $O(n^2)$.

Now let us examine our newly proposed method of detecting conflicts in these trees. The first step is to go through both trees and create summary rules for each tree. Since all rules in both trees need to be traversed, this will take $x + y$ operations. Next 10 comparisons are made between the summary rules to determine if there are any conflicts. If there are no conflicts, we stop after the 10 comparisons. This means the process only took $x + y + 10$ operations. If there are conflicts detected, we move onto the next step.

In the next step, one of the summary rules for R_N's subtree is compared with the rules in R_O. Let us assume the worst case, that the rule in R_O that is causing the conflict will be the last rule traversed in the tree during these comparisons. In that case, finding this rule will take y comparisons. Next, we take this conflicting rule and compare it with the rules in R_N's subtree. Again in the worst case, the conflicting rule in R_N's subtree will be the last rule examined. This will take x comparisons to find. This gives us both of the conflicting rules, with a worst case of taking $x + y + x + y = 2x + 2y$ operations. Assuming x and y are proportional to n, the total number of rules in the system, this process is O(n). Obviously this is a significant improvement over comparing every rule in R_N's subtree with every rule in R_O.

4.4.3. Step 2

Next, we must compare the rules in R_N's subtree with rules found in trees with root R_T, such that R_T and R_O have at least one triggering condition in common. We use the same technique that was described in Step 1 to do Step 2. First we create summary rules for the tree with root R_T. We already have summary rules for R_N from Step 1. We then compare these rules to determine whether there is a conflict present, and if so further analysis is performed to find the two conflicting rules.

In fact, we know that any rule in a tree with root R_T, such that R_O and R_T have one triggering condition in common, cannot conflict with any rule in R_O's tree. This is because R_O's tree and R_T's tree would have been triggered by the same event before R_N was added to the system. Since we assumed that all rules prior to R_N being added were consistent with one another, we know that no rule in the original R_O tree will conflict with any rule in the R_T tree. Therefore we could do step 1 and step 2 at the same time, by compiling all of the rules in R_O's original tree with all the rules in all the R_T's trees, into the same decision table. Then when the process to solve step 1 is executed, it will not only determine whether or not a rule in R_N's subtree conflicts with a rule in the original R_O tree, but also if they conflict with anything in any of the R_T trees. All this is useless, however, if it does not save any time.

Consider if we have R_N such that its subtree has *y* rules in it. Now say that R_O's original tree had *x* rules in it. Also let us assume there are *k-1* trees R_{Ti} that have at least one triggering event in common with R_O. Let's also assume each of these trees has *x* rules. If we compare R_N's subtree to R_O's original tree, and then compare R_N's subtree with each R_{Ti}, we must make *k* comparisons in all. In order to start we must find the summary rules for R_N. This takes *y* operations. Next we must find the summary rules of each R_{Ti}. This takes *kx* operations. For each pair of trees being examined, we must do the following: First we must do 10 comparisons of the summary rules. Next we use Algorithm 1 on R_{Ti}'s tree, taking $C + 2A + 3x$ operations. Finally we use Algorithm 1 on R_N's subtree, taking

$C + 2A + 3y$ operations.

So, for each pair of trees being compared, we take

$10 + 2C + 4A + 3x + 3y$ operations.

So for k pairs of trees being examined, we use

$k * (10 + 2C + 4A + 3x + 3y)$ operations.

If we were to first merge R_O's original tree with all of the R_{Ti}s and then do our comparison we would have the following. Creating the summary rules for R_N's subtree still takes y operations. Creating the summary rules for our new merged tree would take kx operations (note that the new tree has kx rules). Now we need to do 10 comparisons of the summary rules. Next we use Algorithm 1 on our new merged tree, taking

$C + 2A + 3kx$ operations.

Finally we use Algorithm 1 on R_N's subtree taking a maximum of

$C + 2A + 3y$ operations.

So in all we need

$10 + 2C + 4A + 3kx + 3y$ operations.

Therefore it is advantageous to do Step 1 and Step 2 concurrently.

If two rules are found in Step 2 to be conflicting, then we can stop our conflict analysis and report the conflicting rules. Otherwise we must continue on to Step 3.

4.4.4. Step 3

Finally, we must determine if any of the rules in one of R_N's child trees conflicts with another rule in a different child tree of R_N. Since we already have R_{NA+}, R_{NA-}, R_{NO+}, and R_{NO-}, we can compare these with each other to see if anything in R_N's subtree conflicts with anything else in R_N's subtree. To detect modality conflicts we compare:

R_{NA+} and R_{NA-}

R_{NO+} and R_{NO-}

R_{NO+} and R_{NA-}

and to detect action conflicts we compare:

R_{NA+} and R_{NA+}

R_{NA-} and R_{NA-}

R_{NO+} and R_{NO+}

R_{NO-} and R_{NO-}

If no conflicts are detected with the above comparisons, we can say that R_N has not caused the system to become inconsistent. If conflicts have been detected however, then we must find the exact rules causing the conflict. As in Step 1, we take one of the conflicting R_{NX} and use Algorithm 1 to compare it to the rules in R_N's subtree. We will find one of the conflicting rules. Let us call this rule R_{NC1}. Next we use Algorithm 1 to compare R_{NC1} to the rules in R_N's subtree, and we will find R_{NC2}, the rule that conflicts with R_{NC1}.

Note that this algorithm may be performing some unnecessary comparisons. Conflicts can only occur between child trees of R_N if both of the children are triggered by different events. Our algorithm compares all descendants of R_N with all descendants of R_N. We should examine if this is necessary.

Let us compare only the subtrees that need to be examined. In the best case, there will be only two subtrees that need to be examined. In this case only one comparison needs to be made. This comparison involves creating summary rules for each subtree. Assuming each subtree contains *x* rules, this takes *2x* operations. Next 10 comparisons of the summary rules are made. Finally, Algorithm 1 is used twice to find the two conflicting rules, which takes $2 * (C + 2\ (A + x)) = 2C + 4A + 6x$ operations. Therefore the entire process takes $2C + 4A + 6x + 10$ operations.

In the worst case there are *k* subtrees of R_N and all of them have different triggering events, meaning they all need to be compared with each other. Assuming each subtree has the same number of rules, and assuming R_N's subtree has n rules, we can say each subtree has n/k rules. In order to compare all these rules with each other we must compare the first tree with all the others, which is $k-1$ comparisons. The second tree is compared with all the trees except the first tree, which is $k-2$ comparisons, etc., until the second to last tree is compared to the last tree. This takes

$k-1 + k-2 + \ldots + 2 + 1$ comparisons,

which is equal to

$(k-1)*k/2$ comparisons.

From above we know that comparing two trees of size x takes

$2C + 4A + 6x + 10$ operations.

In this case we are doing this $(k-1)*k/2$ times, each time with size n/k. Therefore we have

$$(k-1)*k/2 * (2C + 4A + 6n/k + 10)$$
$$= (k^2/2 - k/2) * (2C + 4A + 6n/k + 10)$$
$$= Ck^2 - Ck + 2Ak^2 - 2Ak + 3nk - 3n + 5k^2 - 5k$$
$$= (k^2 - k) * (C + 2A + 5) + (k - 1) * 3n$$

Now consider doing one comparison of R_N's entire subtree with itself. Worst case and best case would essentially be the same. Either way would take one comparison. This comparison could at worst take

$2C + 4A + 6n + 10$ operations.

Therefore, comparing it to the worst case of comparing every single subtree, for $k = 3$ we see that the previous case takes

$6C + 12A + 6n + 30$,

which is more than the worst case for comparing the entire R_N subtree with itself. Notice that as k increases, the complete revalidation approach takes more and more operations, whereas our method takes a constant number of operations each time.

4.4.5. Analysis

4.4.5.1. Complete Re-validation

If we were to perform a complete re-validation of the system given that it contains X rules, in the worst case we would have to compare the first rule with all other $X-1$ rules, the second rule with the remaining $X-2$ rules, etc. Therefore this would take

$X-1 + X-2 + \ldots + 2 + 1$ rule comparisons.

Assume each rule has c_{avg} conditions and a_{avg} actions. Then each rule comparison would require c_{avg} comparisons of conditions between the two rules, and a_{avg} comparisons of actions between the two rules. This is equal to

$(X) * (X/2) = X^2/2$ rule comparisons * $c_{avg} + a_{avg}$ operations per rule comparison

$= (c_{avg} + a_{avg}) * X^2/2$ operations in total.

Therefore complete re-validation is $O(X^2)$ in the worst case. Note that since we are performing re-validation of the entire system, the best case is the same as the worst case. Therefore the worst case also takes $(c_{avg} + a_{avg}) * X^2/2$ operations, and thus is $O(X^2)$.

In the best case, the first two rules that are compared would conflict and conflict detection would end. This would take 1 operation, and thus is O(1).

4.4.5.2. Algorithm V_2

Let us say that we had the following:

R_N = a rule that is added to a system.

R_O = the root of the tree R_N was added to.

k = the number of other trees R_{Ti} that each have at least one triggering condition in common with R_O.

t = the average number of rules in each R_{Ti}.

y = the number of trees R_N took as its children when it was added.

s = the size of the R_N new child trees.

n = the number of rules in R_N's tree.

m = the number of rules in R_O's original tree.

In section 4.4.3 we showed that merging R_O's original tree with all of the R_{Ti}s to compare with R_N's subtree takes

$10 + 2C + 4A + 3(kt+m) + 3n$ operations in the worst case.

In 4.4.4 we showed that Step 3 takes

$2C + 4A + 6n + 10$ operations in the worst case.

Therefore our entire process in the worst case takes

$10 + 2C + 4A + 3kt + 3m + 3n + 2C + 4A + 6n + 10$

$= 20 + 4C + 8A + 3kt + 3m + 9n$ operations.

Assuming m, n, and t are proportional to X, we can say that in the worst case our algorithm is O(X).

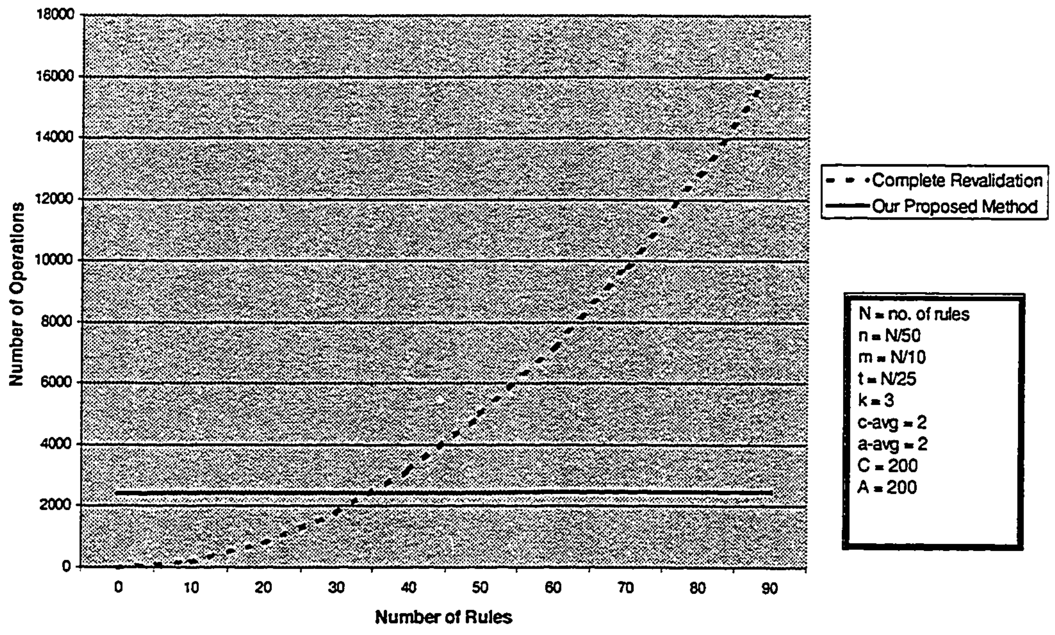

Figure 17 - A Graphical Comparison of the Two Algorithms

In the best case Step 1 and 2 are completed without detecting a conflict. It takes $kt + m$ operations to compile the summary rules for the merged tree. Next it takes n operations to compile the summary rules for R_N's subtree. Finally 10 comparisons are made and it is determined there are no conflicts. Similarly in Step 3 we do 10 comparisons of R_N's summary rules, and find no conflict. In this case, the entire process takes *20 + kt + m* operations. If once again we assume that m and t are proportional to X, we can say that in the best case our algorithm is O(X). Figure 17 shows a graphical comparison of the complete revalidation method with our algorithm. Note that because of the large difference in performance the scale of the graph was chosen to illustrate that complete revalidation performs in quadratic time. This scale gives the illusion that our method performs in constant time, when in fact it performs in linear time with a small slope.

4.5. Cyclic Conflict Detection

Another possibility for a conflict is if a rule R_I triggers a rule R_J which triggers a rule R_T etc., until finally a rule R_T is triggered which triggers R_I again. This causes a cycle of triggers. This means that rules are constantly being triggered, and may never stop executing. Not only this but depending on the rules being triggered, if a job is being passed onto the next rule, then the job will never be completed because no rule ever tackles the problem. For example say R_1 owned by Bill states that upon receiving a work-related e-mail, the e-mail should be forwarded to John, and R_2 owned by John states that upon receiving a work-related e-mail, the e-mail should be forwarded to Bill. In this case if either Bill or John ever receives a work-related e-mail one rule will trigger the other which triggers the first rule, and this cycle continues on and on and on. The e-mail never gets read, and many CPU cycles are wasted by from this endless passing the buck. This behaviour needs to be detected in the rules and reported.

Let us make a graph showing each rule as a vertex in this graph. Now let us connect the vertices using directed edges, each edge drawn from one vertex to another representing that the first rule triggers the second. Now if we examine the graph and detect a cycle, we know that one rule may, after a trigger chain reaction, effectively trigger itself.

In Figure 18 we see that R_1 triggers R_3 which triggers R_6 which triggers R_1. Since this forms a cycle in our directed graph, we know that there is a potential cyclic conflict involving R_1.

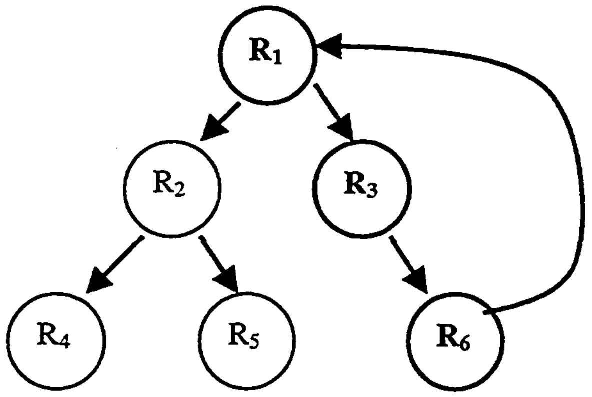

Figure 18 – A Cyclic Conflict

Note that if we were to draw a similar graph, this time having the edges representing if a rule potentially triggers another rule, and performed the same search for cycles, we could find potential conflicts of this sort. Since, however, the chance of all the conditions in one of these cycles being met over and over, without a rule to throw all the events needed, is not particularly high in most cases, this is less likely to result in a conflict. Still, because of the potential conflict, it is a pattern that should be detected so that at very least the user can be warned, and in the best case the policy will be redesigned.

4.6. Summary

In summary, although the best case of the complete revalidation approach performs better than our algorithm in the best case, our algorithm performs much better in the worst case. Also our best case (having no conflicts) seems much more likely to occur than having the two first rules in the system to conflict, as is needed for the best case of the complete revalidation approach. Regardless of this, we have shown considerable improvement over the complete revalidation approach with our algorithm, and thus have demonstrated the benefits of incremental validation.

5. Architecture for Incremental Validation

We have shown how incremental policy validation can improve the performance of policy-based systems. It is now important to show how policy-based systems can be built that can use incremental validation. To do this, we will take an existing policy architecture and show what modifications will be necessary in order to take advantage of incremental validation.

5.1. An Existing Architecture

In his thesis, Kanthimathinathan [42] described an architecture that supports many popular policy models. The main modules necessary for policy input are a PSL Editor, an EA Model Handler, a Policy Handler, a Conflict Handler, and a Policy Engine, as illustrated in Figure 19. A policy is entered into the system through the PSL Editor. From there it is sent to the EA Model Handler where it is examined to see if the specifier of the policy has the permissions necessary to act upon the targets and other program entities indicated in the policy. If this level of verification succeeds, the policy is then sent to the Conflict Handler module. Kanthimathinathan does not mention the explicit purpose of the Conflict Handler but presumably it is to check for such things as modality conflicts and application specific conflicts, since after this stage the policy is sent to the Policy Service where it begins its transformation into executable rules. These executable rules are then sent to the Policy Engine which enforces them.

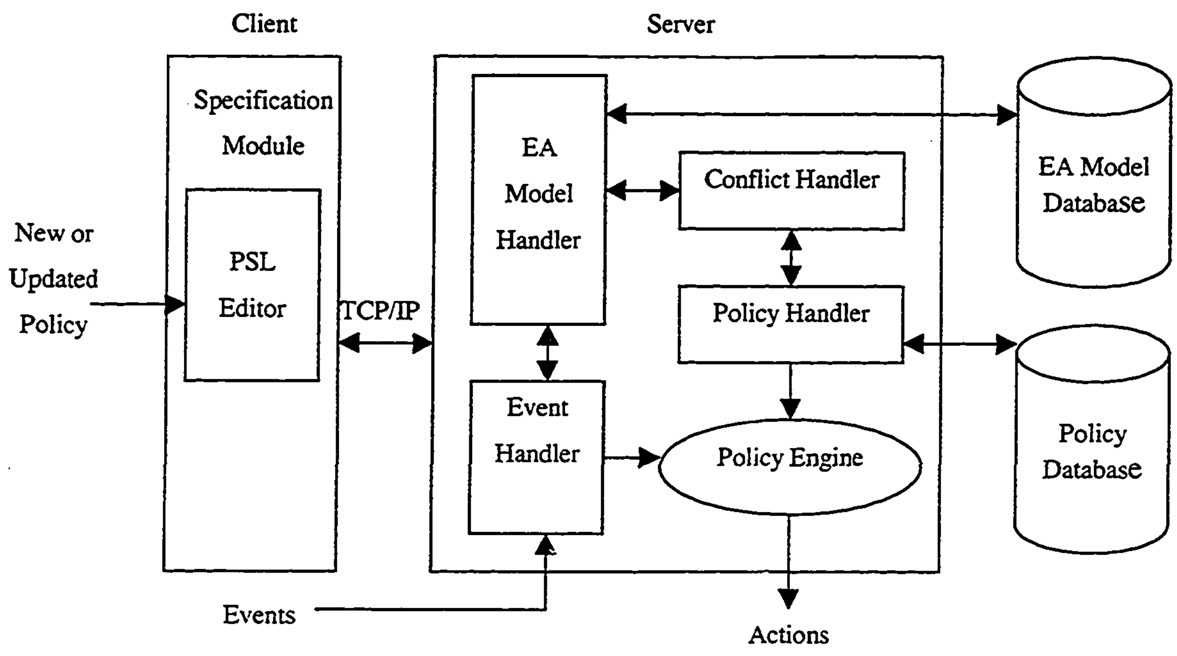

Figure 19 – Kanthimathinathan's Policy Architecture

5.2. What is Needed in a Conflict Handler

We will now define what should be present in this Conflict Handler module. Since we can add policies to the system one at a time, we can perform incremental validation as each policy is added. Then we may think it is possible to always validate the policies using the incremental validation technique. However, there are cases where a validation of the entire set of policies is necessary. Take for example the case where an event is removed from the system. There could be two rules that have identical condition sets. Since these rules had overlapping condition sets to begin with, they already would have potentially conflicted, so we do not need to perform a full policy set validation due to this. However, these rules could now become identical. This is not really a problem, but having multiple identical rules takes up unnecessary space and processing time during validation, so these duplicate rules should be removed.

We could also have rules which have empty condition sets. In this case, these rules should be removed from the system. Therefore every time an event is removed from the system, all the rules in the system should be examined and all rules with empty condition sets should be removed. The same is true if an object that is tested in condition statements is removed from the system. Although the entire policy set must be examined, this scenario does not introduce any conflicts into the system, therefore it is not necessary to perform any conflict detection between policies.

If a target or a function that can be performed on a target is removed from the system, then we have a similar case to the case above. We could have two or more rules that now have identical action sets. This does not pose a problem since we are allowing multiple rules to have identical actions sets. However, again, if these rules are identical then they should be removed to reduce the amount of resources needed. Just like removing an event could result in rules with empty condition sets, removing a target or target function can result in rules with empty action sets. Therefore every time a target or target function is removed from the system, all the rules in the system should be examined and all rules with empty action sets should be removed. Again in this scenario no conflicts between policies are introduced into the system and hence conflict detection between policies is not necessary in this case.

If an event, target, or function that can be performed on a target is added to the system, there is almost no change in the system. The decision tables in the system will have to have another row added to them, which will result in a new entry for each rule. By default this entry will be blank which represents a don't-care-entry, accepting all values for the new attribute. We can see then, that the set of policies will not change in terms of what events and conditions trigger them, and what actions they perform. Therefore, when a new event, target, or function that can be performed on a target is added to the system, no re-validation of the set of policies is necessary.

When a new policy is added to the policy set, or an existing policy is modified, we have the possibility of introducing conflicts into the system. As we have shown in

Chapter 4, these conflicts can be detected efficiently using an incremental validation technique. As long as incremental validation is performed after every policy addition or modification, no method of validating the entire set of policies will be needed for such modifications to the system.

Therefore, from the above case analysis, we can see that our Conflict Handler will need two types of policy validation. The first is the incremental conflict detection explained in Chapter 4. The second is the ability to examine all the policies for duplicate or empty rules and remove them. If we include trigger chaining as a property we can check for, then cyclic conflict detection should also be performed.

In order to perform incremental validation and cyclic conflict detection our Conflict Handler can use a tree representation of the set of policies.

5.3. Scenarios

This section provides various scenarios for changing the policy set in the system. We discuss what the flow of information when a policy is added, modified, or deleted from the system, and what happens when conditions, events, and actions are added or removed from the system.

5.3.1. Adding a policy

Upon adding a policy to the system, the new policy will move from the PSL Editor to the EA Handler. From here it will be sent to the Conflict Handler. The Conflict Handler will then create a decision table of all policies that are in the same scope as the new policy including the new policy itself. The policy will also be added to the appropriate place in the policy tree. The tree will then be used to perform incremental validation on the policy. If validation fails, the policy will be removed from the policy tree. If validation succeeds, then the policy will be added to the appropriate decision table, and the policy will be sent to the Policy Service where it can be translated into executable rules and sent to a Policy Engine.

5.3.2. Modifying an Existing Policy

The scenario for modifying an existing policy is nearly identical to adding a new policy. In this case, when performing validation, one proceeds as indicated in 5.3.1, however, this time the policy that the new policy is replacing should be excluded from the decision table and the policy tree. If validation succeeds, the new policy will replace the old policy both in the decision table and policy tree. The new policy is sent to the Policy Service which translates the policy into executable rules which are sent to the Policy Engine. The Policy Service also requests the old policy be removed from the Policy Engine.

5.3.3. Deleting a Policy

In the case of deleting a policy, there is no conflict detection to be done, provided that no policies are dependent on other policies. In this case, the Conflict Handler can be completely bypassed and the Policy Service will request the indicated policy be removed from the Policy Engine.

5.3.4. Adding an Event, Condition, or Action

If an event, condition, or action is added to the system, then all decision tables in the system must be modified to accommodate this change. This is done by adding a row with the appropriate stub to all decision tables in the system. This will not result in any conflicts, so the Conflict Handler is not needed in this case.

5.3.5. Deleting an Event, Condition, or Action

When an event, condition, or action is deleted from the system, the row containing that attribute must be deleted from every decision table in the system. This could result in duplicate rules, or rules that have completely empty condition entries or action entries. In the case of duplicate rules, this does not lead to a conflict, however, duplicate rules are unnecessary and take up an extra amount of memory and processing time. This is especially true when validating rules, which we have shown to be dependent on the number of rules in the system. Therefore the rules should be examined and any duplicate rules should be removed. Similarly, if there are rules with empty condition entries or action entries, these rules are not valid, and should be removed. Therefore, in this case, the Conflict Handler should modify all decision tables in the system and remove any duplicate or empty rules. The policy tree does not need to be modified in this case. The Conflict Handler should then notify the Policy Handler which rules should be updated or deleted from the Policy Engine.

5.4. Transition to Enforcing Updated Policies

When new policies are added to the system, the objects using the system must be updated to use the new policies. An important question, however, is exactly how the transition will be made to the new policies. We suggest two ways to do this: the stop and reload method, and the static generation method.

5.4.1. Stop and Reload Method

The idea behind the stop and reload method is to provide all the objects in the system with the very latest updates to the policies. In order to do this, the execution of the system should be stopped, so that all objects can update their policies from the database. The advantage of this method, of course, is that all objects will have the very latest version of their policies. This also means that all objects in the system will have the same version of a particular policy and so all objects referring to a particular policy will behave in the same manner. The disadvantage to using this technique is that the system must be halted temporarily while the objects refresh their policies. Depending on the number of objects and policies in the system, this may take a noticeable amount of time and may disrupt the appearance of continuous service to the users of the system.

5.4.2. Static Object-Policies Method

The stop and reload method of policy transition required that the system halts execution while the objects refresh their policies from the database. The static object-policies method avoids this. With the static object-policies method, each object fetches its policies from the database as the object is created and never updates its policies again. If a change is made to the policy database, the change will affect the new generation of objects, but the objects created before the policy change will retain the old version of the policy. This solves the problem of having to halt the system every time a change is made to the policy database. However, it does have disadvantages.

One disadvantage is that there are various objects in the system which have differing versions of the same policy. This means that otherwise identical objects may behave differently because of the varying policy versions. The other disadvantage is that the policies for each generation of objects is static, and so although the policy-based system may be dynamic, i.e., it allows policies to be changed during the execution of the system, the objects will never inherit these new behaviours. The objects will only take on the new behaviours if the objects are deleted and created again.

5.5. Summary

Building a system which implements incremental policy validation is not a simple task. However, extending the architecture of a current policy-based system makes this task somewhat easier. We have shown the components that are required to make an existing policy-based system able to validate its policies incrementally. We have also discussed the problem of policy transition when policies are changed in the system, and offered two possible solutions to this problem.

6. Conclusion

The main contribution of this thesis was exploring the problem of incremental policy validation, and providing a method of solving this problem. Incremental policy validation is an area in which no prior work has been done so far. With the increase in number of systems that use policies, and the large size of policy systems, incremental policy validation will become an important problem. By validating only those rules which have a chance of conflicting, we can substantially reduce the execution time of the validation process of policy systems.

As a way of solving this problem, we have developed an algorithm. We have developed the concepts used in the algorithm, in a step by step manner, and refined those concepts to arrive at the final algorithm. Then, we analyzed the algorithm for its performance and compared it to the complete revalidation approach of revalidating all policies in the system. This analysis showed that incremental policy validation does offer a significant advantage over complete revalidation.

Trigger chaining was also introduced in this thesis. Trigger chaining consists of looking at policies to see which chain of policies they trigger. This allows us to find more conflicts, and so an incremental validation algorithm was suggested to find these conflicts. This algorithm was analysed and compared to a complete revalidation method. Our incremental validation method was shown to operate in linear time, as opposed to the complete revalidation method which was found to have quadratic complexity.

Trigger chaining also introduced a second new type of conflict. Cyclic conflicts were introduced as a policy which triggers a chain of policies to fire, resulting eventually in its own execution. This new type of conflict was not examined in much detail in this thesis. However, it was suggested that directed graphs could be used to detect these conflicts.

We have also introduced the idea of representing policies in decision tables. There has been much work done in the past related to decision tables which we can use with respect to policies. The problem of decision table completeness and decision table consistency have all been studied in the past. They helped us in refining the concepts in our current approach to policy validation, and they can also help in some future work. The problem of converting decision tables into executable programs has also been examined at length by other researchers. This research can be used to solve the problem of converting policies to some form of executable code, which needs further exploration.

Decision tables have also given us a benefit in the form of conceptual simplicity. By representing policies in this simple representation, it is easy to see the differences between policies, what happens when new policies are added or old policies are removed, etc. This view makes it simpler to develop algorithms related to policies. In fact, this representation helped in the task of developing an incremental validation algorithm, by providing a simple way to eliminate conditions, actions, and entire rules from the search space. We presented policies in the form of decision tables and then showed the limitations of this format. We then extended the decision table format to meet the requirements of representing policies.

Developing a system using incremental policy validation is not a small task. We have taken an existing policy framework from the literature and have shown what modifications are necessary in order to achieve a system capable of incremental validation.

7. Future Work

In this thesis we have shown that the concept of incremental policy validation is a valid one, and is useful. We have introduced an algorithm which solves this problem and performs better than the complete revalidation approach. Future work may include developing and analyzing an algorithm which performs the same task but does it with optimum performance.

Further work could also include implementing the incremental algorithm given in this thesis to show that it performs well in practice as well as in theory. This would ideally be done with a large system consisting of many rules, since as we have shown, the difference between the two algorithms becomes very apparent when there are a large number of rules in the system.

Although we did not explicitly restrict our algorithm to simple decision tables, whereby each entry has only yes or no values, we did not go into detail how tables with allowing entries with multiple values would affect incremental policy validation. Similarly we did not examine the case of two conditions that overlap. For example, in our work we have considered conditions such as "(x = 6)". In our system a rule with condition "(x = 6)" would not conflict with a rule with condition "(x > 3)" since the two conditions are not the same. Our table would have a row in the decision table for each condition and our validation algorithm only checks if two rules are triggered by the same row in the decision table. If we were to allow conditions such as "x" where the values could be " = 6" or " > 3", then we could check for conflicts more completely.

Trigger chaining was introduced as a concept in our thesis, but it was only used to examine incremental static policy validation. Trigger chaining may or may not have an effect on run-time policy validation, which remains to be explored. Similarly, cyclic conflict detection was also introduced, but was not examined in very much detail. It would be useful to see if cyclic conflict can occur at run-time provided an incremental static policy validation mechanism is in place. If so, it would be worth while to develop an algorithm to solve this problem.

We introduced the notion of representing policies as decision tables in this thesis. This is one area that could possibly be expanded upon in the future. Benefits taken from past work in the area of decision tables could be examined to see if it can be used to improve policy systems using this representation. For example, the notion of intracolumn inconsistency may aid in creating a more efficient validation algorithm. Also, the extended decision table format we introduced could possibly be expanded upon to add functionality which our format does not currently provide. For example, the decision table itself does not currently indicate who is able to change the policies or where the policies sit in the organizational hierarchy.

8. References

[1] Gruber, R. E., Krishnamurthy, B., Panagos, E., "High-Level Constructs in the READY Event Notification System", *Eighth ACM SIGOPS European Workshop*, Sinatra, Portugal, Sept. 1998.

[2] Krishnamurthy, B., Rosenblum, D. S., "Yeast: A General Purpose Event-Action System", *IEEE Transactions on Software Engineering*, vol. 21, no. 10, Oct. 1995.

[3] Rosenblum, D. S., Wolf, A. L., "A Design Framework for Internet-Scale Event Observation and Notification", *Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT symposium on Software engineering*, September 22 - 25, 1997, Zurich Switzerland, pp. 344-360

[4] Lupu, E., Sloman, E., "Conflicts in Policy-Based Distributed Systems Management", *IEEE Transactions on Software Engineering* vol. 25, no.6. Nov./Dec. 1999.

[5] Omari, S., Boutaba, R., "Policy-Based Control Agents for Boundary Routers in Differentiated Services IP", *First International Workshop on Mobile Agents for Telecommunications Applications*, pp. 477-490, Oct. 1999.

[6] Lupu, E., Sloman, E., "Conflict Analysis for Management Policies", *Proceedings of the 5th International Symposium on Integrated Network Management IM'97*, 1997.

[7] Cholvy, L., Cuppens, F., "Analyzing Consistency of Security Policies", *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp.103-112, 1997.

[8] Sommerville, I., Software Engineering, Fifth Ed., Addison-Wesley, USA, 1996.

[9] Michael, J., Sibley, E., Littman, D., "Integration of Formal and Heuristic Reasoning as a Basis for Testing and Debugging Computer Security Policy", *Proceedings of the New Security Paradigms Workshop*, Los Alamitos, California, pp. 69-75, 1993.

[10] Moffett, J. D., Sloman, M. S., "Policy Hierarchies for Distributed Systems Management", *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 9, pp. 1404-1414, Dec. 1993.

[11] Grosof, B. N., Labrou, Y., Chan, H. Y., "A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML.", *Proceedings of the First ACM Conference on Electronic Commerce*, Nov. 1999.

[12] Chomicki, J., Lobo, J., Naqvi, S., "Conflict Resolution in Policy Management", www.cs.buffalo.edu/~chomicki/papers-tkde01.ps, Submitted June 12, 2000

[13] Smith, I. A., Cohen, P. R., Bradshaw, J. M., Greaves, M., Holmback, H., "Designing Conversation Policies Using Joint Intention Theory", *Proceedings of the International Conference on Multi Agent Systems*, pp. 269-276, 1998.

[14] Grossner, C., He, X., Kurusetty, B., Mahoney, G., Radhakrishnan, T., "The Use of a Restricted Natural Language and an Intermediate Representation in Policy Based Systems", Work in Progress

[15] Falchuk, B., Karmouch, A., "Visual Modeling for Agent-Based Applications", *IEEE Computer*, Dec. 1998, pp. 31-38.

[16] Jajodia, S., Samarati, P., Subrahmanian, V. S., "A Logical Language for Expressing Authorizations", *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pp. 31-42, 1997.

[17] Koch, T., Krell, C., Krämer, B., "Policy Definition Language for Automated Management of Distributed Systems", *Proceedings of the Second IEEE International Workshop on Systems Management*, pp. 55-64, 1996.

[18] Cuppens, F., Cholvy, L., Saurel, C., Carrere, J., "Merging Security Policies: Analysis of a Practical Example", *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pp.123-136, 1998.

[19] Howard, S., Lutfiyya, H., Katchabaw, M., Bauer, M., "Supporting Dynamic Policy Change Using CORBA System Management Facilities", *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management (IM '97)*, pp. 527-538, 1997.

[20] Cuppens, F., Saurel, C., "Specifying a Security Policy: A Case Study", *Proceedings of the 9th IEEE Computer Security Foundations Workshop*, pp.123-134, 1996.

[21] Bertino, E., Jajodia, S., Samarati, P., "Supporting Multiple Access Control Policies in Database Systems", *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp.94-107, 1996.

[22] Howard, S., Lutfiyya, H., Katchabaw, M., Bauer, M., "Supporting Dynamic Policy Change Using CORBA System Management Facilities", *Proceedings of the Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM'97)*, San Diego, California, May 12-16, 1997.

[23] Fraser, T., Badger, L., "Ensuring Continuity During Dynamic Security Policy Reconfiguration in DTE", *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, 1998.

[24] Mahon, H., Bernet, Y., Herzog, S., Schnizlein, J., "Requirements for a Policy Management System", IETF Internet Draft, www.ietf.org/internet-drafts/draft-ietf-policy-req-02.txt, Nov. 9, 2000.

[25] Shay, W. A., Understanding Data Communications and Networks, PWS Publishing Company, USA, 1995.

[26] Moore, B., Ellesson, E., Strassner, J., Westerinen, A., "Policy Core Information Model", IETF document RFC 3060, www.ietf.org/rfc/rfc3060.txt, Feb. 2001.

[27] Hilpinen, R. (ed.), Deontic Logic: Introductory and Systematic Readings, D. Reidel, Dordrecht-Holland, 1971.

[28] Marriott, D., Sloman, M., "Implementation of a Management Agent for Interpreting Obligation Policy", *IEEE/IFIP Workshop on Distributed Systems Operations and Management (DSOM '96)*, Laquila, Italy, Oct 1996.

[29] Montalbano, M., Decision Tables, Science Research Associates, USA, 1974.

[30] Welland, R., Decision Tables and Computer Programming, Hyden & Son, Great Britain, 1981.

[31] Ibramsha, M., Rajaraman, V., "Detection of Logical Errors in Decision Table Programs", Communications of the ACM, vol. 21, no. 12, pp. 1016-1025, Dec. 1978.

[32] King, P. J. H., Johnson, R. G., "Some Comments on the Use of Ambiguous Decision Tables and Their Conversion to Computer Programs", Communications of the ACM, vol. 16, no. 5, pp. 287-290, May 1973.

[33] Shwayder, K., "Combining Decision Rules in a Decision Table", Communications of the ACM, vol. 18, no. 8, pp. 476-480, Aug. 1975.

[34] Muthukrishnan, C. R., Rajaraman, V., "On the Conversion of Decision Tables to Computer Programs", Communications of the ACM, vol. 13, no. 6, pp. 347-351, June 1970.

[35] Lew, A., "Optimal Conversion of Extended-Entry Decision Tables with General Cost Criteria", Communications of the ACM, vol. 21, no. 4, pp. 269-279, April 1978.

[36] Shumacher, H., Sevcik, K.C., "The Synthetic Approach to Decision Table Conversion", Communications of the ACM, vol. 19, no. 6, pp. 343-351, June 1976.

[37] Dathe, G., "Conversion of Decision Tables by Rule Mask Method Without Rule Mask", Communications of the ACM, vol. 15, no. 10, pp. 906-909, Oct. 1972.

[38] Giarratano, J., Riley, G., Expert Systems Principles and Programming, PWS Publishing, USA, 1998.

[39] Jess the Java Expert System Shell, http://herzberg.ca.sandia.gov/jess; accessed March 12, 2001.

[40] Rowe, N. C., Artificial Intelligence Through Prolog, Prentice Hall, USA, 1998.

[41] Vanthienen, J., Wets, G., "From Decision Tables to Expert System Shells", Data & Knowledge Engineering 13(3), pp. 265-282, Oct. 1994.

[42] Kanthimathinathan, V., "An Enterprise Policy Specification Tool", M. Comp. Sci. Thesis, Concordia University, Feb. 2001.