FlexFlow: A Flexible Flow Control Policy Specification Framework*

Shiping Chen, Duminda Wijesekera and Sushil Jajodia
Center for Secure Information Systems
George Mason University, Fairfax, VA 22030
e-mail: {schen3|dwijesek|jajodia}@gmu.edu

Abstract

Flow control policies are important in data-flow, work-flow, transaction systems and software design. Previous work in this area concentrates either on modelling security aspects of information flow control or applying flow control policies in some specific application domain. These models permit either permissions or prohibitions for flows and normally are based on a specific meta-policy (usually the *closed* policy).

We propose FlexFlow, a logic based flexible flow control framework to specify data-flow, work-flow and transaction systems policies that go beyond point-to-point flows. Both permissions and prohibitions are specifiable in FlexFlow and meta-policies such as *permissions take precedence* themselves can be specified over the meta-policy neutral policy specification environment of FlexFlow. We further show how to specify and prevent inter-flow conflicts such as those arising in role-based work-flow policies.

Key Phrases: Flow control policy, Data flow, work flow, security policy, security constraints.

1 Introduction

Information flow policies govern the exchange of information at various levels in systems. At the lowest levels, information is copied in and out of registers and memory locations inside processors. At a higher level, information is exchanged among variables in programs, methods in object oriented systems and transactions in database systems. In networked systems, messages are copied across system boundaries in order to exchange information. In all of these examples the levels at which information flows, the units of transfer and the number of destinations vary, but the central issue of information flow remains the same. Thus, it is interesting to investigate the commonalities among policies that govern information flow at an abstract level. This paper proposes FlexFlow, a framework to do so.

Being designed to capture properties common across flows, FlexFlow is formulated using abstractions of *nodes* and *trees* of flows among them. Because of this abstractness, FlexFlow has two advantages. Firstly, FlexFlow is not limited to high or low level information exchange policies. Thereby it can be used to reason about and derive consequence of mixing flow control polices at different levels. For example, higher level information exchange policies may govern flows between method calls, and lower level policies may govern data flows inside method calls. Because our framework can model information flow at both levels, it is able to combine policies across both of them.

The second advantage is that our policy specification framework does not depend on any meta policies. Therefore, policies using different meta policies can be modelled and their total effect can be compared using our framework. This is similar to the advantage gained by the *Flexible Authorization Framework* (FAF) of Jajodia et al. [JSSS01] over its predecessors in specifying access control policies.

FlexFlow is similar to FAF in many other ways. Firstly, FlexFlow specifies flow control polices as a set of stratified Horn clauses. Secondly, FlexFlow is sound and complete, in the sense that every requested flow is either granted or denied, but not both. Thirdly, FlexFlow is also based on unique stable model semantics, and therefore FlexFlow rules can be interpreted unambiguously. Fourthly, because Horn clauses can be used to derive new Horn clauses, the back tracking procedures used to execute query against FlexFlow can also be significantly optimized using well known techniques in logic programming. Lastly, as described in [JSSS01], FlexFlow polices can also be materialized, thereby saving query execution times during flow control requests.

*This work was partially supported by the National Science Foundation under grant CCR-0113515.

Meanwhile, FlexFlow is different from FAF [JSSS01] in some aspects. The main difference is in their logical formulation. FlexFlow use lists, whereas FAF does not. Therefore, unlike many logic based formulations of policies, FlexFlow is set-based as opposed to entity-based. Although this paper does not show in detail, this logical formulation with list processing capability can be used to specify and resolve many application dependent conflicts. Such method has been used in Authorization Constraints Specification in work-flow Management System in the past [BA99]. Additionally, because basic objects of study in FlexFlow are trees, inter-flow and intra-flow conflicts can be specified and resolved. Although not fully worked out in this paper, to the best of our knowledge, this subtle difference does not exist in FAF and previous formulations of logic based flow control frameworks. Additionally, the tree building capability of FlexFlow can be optimized by using standard tree manipulating algorithms. Our ongoing work address these issue.

The rest of the paper is organized as follows. Section 2 informally introduces abstract concepts of nodes, flow trees and their representations. Section 3 formally describes the FlexFlow framework. Section 4 shows that FlexFlow can be used to express various existing flow control models. Section 5 addresses flow constraints in FlexFlow. Section 6 describes related work and Section 7 concludes the paper. Appendices A and A.1 provide some auxiliary details necessary to model database transactions in FlexFlow.

2 An Informal Description of FlexFlow

This section informally describes the FlexFlow framework and the reasons that went into making our design choices. FlexFlow has trees referred to as *flow trees* build up from *nodes* and *branches*. Nodes represent sources and sinks of information and branches represent pathways taken by information flowing between nodes. Thus, information flows from the leaves of a tree via intermediate nodes to its root. Given that the same node can either send information or refuse to do so under different circumstances, FlexFlow uses *node environments* to capture sufficient data to enforce such *local* decisions. Thus node environments are used to specify and enforce information filtering and mixing policies at nodes of flow trees. Similarly, a flow tree itself may be acceptable or rejectable due to policies under different global circumstances, despite the nodes enforcing their local policies. Such circumstances are captured by tree environments of a flow tree. The exact relationship between the environment of a flow tree and those of its nodes is not rigidly fixed by FlexFlow and is therefore application specifiable. Consequently, the flow trees with their environments, consisting of nodes and node environments constitute the basic entities of focus in FlexFlow. In order to specify how to construct acceptable or rejectable flow trees, FlexFlow has rules written in the form of Horn clauses about trees and their structural properties. Our position - one that we put forth in this paper - is that such rules suffices to enforce existing flow control policies in a uniform and meta-policy independent manner.

As an example, consider an abstract syntax tree of the expression `x+(y+z)`. As shown in the left hand side of Figure 1 (the right hand side is its Prolog representation - to be explained shortly), it consists of three leaves with variables (variables are considered named locations that can hold values) `x`, `y` and `z`, one intermediate node R_{temp} and a root node R_{final}. Assume that each variable in this example is accessible by a specified set of subjects. For example, `x` is accessible by Alice and Bob, `y` accessible by Bob and Cindy, and `z` accessible by Alice, Bob and Cindy. We model this situation by making the environment of each node be the access control list. We view each binary addition operation as a computation tree in which the root stores the sum of the values stored in the leaves. Thus, as shown in Figure 1 there is an *intermediate* node R_{temp} holding the value of `x+y` and the root R_{final} holding `x+(y+z)`. As it is shown in Figure 1, R_{final} holds the value of `x+`R_{temp}. The tree rooted at R_{temp} is said to resemble a *one step* flow, as it has depth one. By piecing together two trees with depth one, (namely the depth-one binary tree rooted at R_{final} and the one rooted at R_{temp}) we get a depth-two flow tree, namely the tree rooted at R_{final} with `x`, `y` and `z` as leaves.

Having to reason with tree structures and lists in Horn clauses, FlexFlow uses Prolog's list notation to represent trees. In that notation, a list consisting of A, B and C are represented as [A,B,C]. Furthermore in the same notation, a tree is represented as a list where the first element is the root of the tree and other elements are the subtrees rooted in the left to right order. For example, [1,[[2,[3,4]],5]] represent a tree with 1 as its root and [2,[3,4]] and 5 as its two children. The first child [2,[3,4]] itself is a root of a tree with two children, namely 3 and 4. The second child of 1 in the original tree is 5. Thus, as can be seen from this example, in using the Prolog notation for lists to represent trees, nodes are explicitly stated, but branches connecting them are implicit and derived from the nesting of lists (i.e. square brackets). We use this notation for trees in FlexFlow.

In FlexFlow, we use ordered pairs of (node names, environments) as our nodes. In list notation, these are repre-

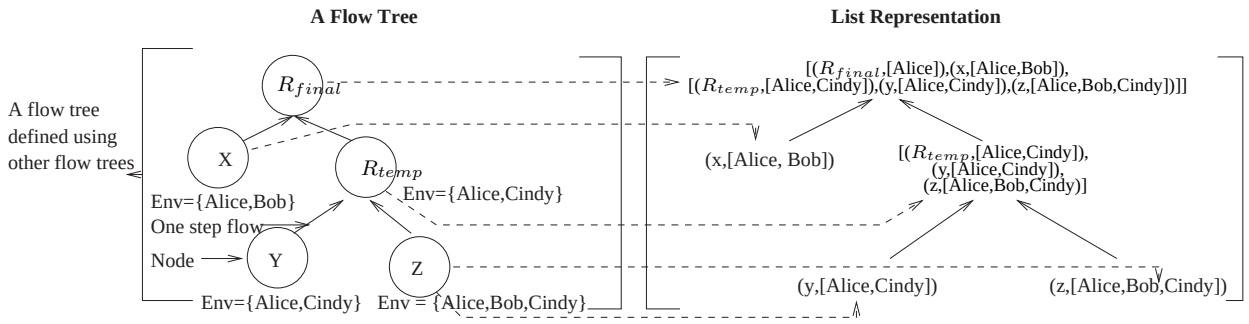

Figure 1: An Example Flow Tree

sented by [node name, environments], which we sometimes write as (node names, environments) in order to avoid being lost between square brackets. Using that convention, `(z,[Alice,Bob,Cindy])` represents the rightmost bottom (i.e. the last in the in-order representation of the tree) node in Figure 1. Therefore, using our convention to represent nodes and Prolog's list notation, the subtree rooted at R_{temp} with an access control list [Alice,Cindy] and two children `y` and `z` is represented as `[(`R_{temp}`,[Alice,Cindy]), [(y,[Alice,Cindy]), (z,[Alice,Bob,Cindy])]]`.

FlexFlow states flow control policies as Horn clauses using flow trees as individual elements. In order to create these rules we used some predicates. Both rules and predicates used in FlexFlow belong to a five level stratification. Predicates at lower strata allow the construction of one step flows. At the higher levels, these one step flows are used to construct flow trees. The stratified rules in FlexFlow can be used to specify flows that are permitted or prohibited without assuming any meta policies such as the open or closed policy. Formal details are given in Section 3.

FlexFlow has made several design choices. The first one is to build flow trees instead of maintaining associations of sources with their respective destinations. One advantage of building trees as opposed to the latter alternative is that trees contain entire transcripts of flow histories, and therefore can be used to specify fine-grained flow control policies. That requires policies that controls flows locally as well as globally. For example, as will be shown later, FlexFlow could accept the computation `x+(y+z)`, but not `(x+y+z)`. Notice that in order to do so, the policy rules must not permit an intermediate computation of `(x+y)`. This can be specially useful in specifying unwanted conflicts that may occur inside a flow tree. For example, we may reject computing `x+y` if two subjects in the access control lists of `x` and `y` play conflicting roles, such as one producing data and the other consuming them. Maintaining only source lists and destinations do not allow us to specify this fine-grained difference that occur in intermediate nodes of flow trees. We now describe FlexFlow architecture.

2.1 Architecture of FlexFlow

As shown in Figure 2, FlexFlow rules have five stages, numbered zero through four in the figure. The first of them is the structural module (stage 0) containing data structures and functions needed to define flows. These include subject and object hierarchies, predicates necessary to model the environments and list manipulation operations such as adding or removing elements, taking the union of two lists etc.

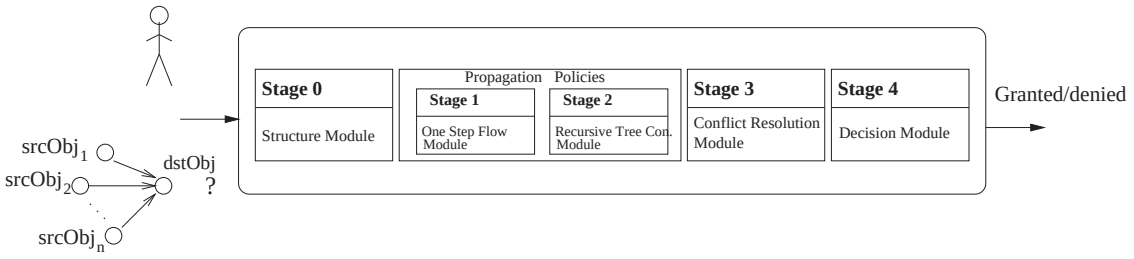

Figure 2: FlexFlow System Architecture

Stage 1 is used to specify one-step flows that are permitted or prohibited based on the structural properties specified

at the previous stage. As show in Figure 1, this stage consists of constructing the tree rooted at node R_{temp}. The next, stage 2, is used to build the permitted or prohibited flow trees that may use already defined one-step flows. Recursion is allowed in this step. For example if flows are transitive, they can be specified at this stage, as transitive closure can be recursively defined. Similarly, permissions can also be propagated up and down subject, object and role hierarchies using recursive rules.

Although propagation policies are flexible and expressive, they may result in *over specification*. That is, rules could be used to derive both negative and positive flows that may be contradictory. This possible conflict is due to the fact that positive and negative permissions is an application level inconsistency is not recognized by the underlying stable model semantics of locally stratified logic programs. Similar encodings have been used in the *Flexible Authorization Framework* by Jajodia et al. [JSSS01]. In order to weed out contradictive specifications, FlexFlow uses *conflict resolution policies*. They are stated in stage 3.

At stage 4, *decision policies* are applied in order to ensure the completeness of flow specification. That is because every flow request made to FlexFlow must be either granted or denied. This is necessary because, as otherwise the framework makes no assumption about flows that are not derivable only using stated rules.

3 Syntax and Semantics of FlexFlow

3.1 The Language of FlexFlow

Terms of the language: The language of FlexFlow consists of terms that are made up from constants and variables for nodes, environments and actions. It also has constants and variables over lists of (node, environment) pairs. Such lists are considered as flow trees, where the first element is the root and the rest are the children. In addition, FlexFlow allows environments to have application defined structures and predicates. An example of such a structure is an access control list, as used in Figure 1.

Predicates of the language: Predicates in FlexFlow belong to five strata, as summarized in Table 1 and explained below.

Stratum 0: Consists of list manipulation and application-specific predicates. An example of an application specific predicate is `playRole(`x_s, x_r`)` stating that subject x_s plays role x_r. An example list manipulation predicate is `isMember(`$(x_n, x_e), X_L$`)` stating that the (node, environment) pair (x_n, x_e) is in the list X_L. More examples are given in section 4.

Stratum 1: Consists of a four-ary predicate safeFlow. The formal parameters x_n, x_e, X_L and x_{action} of safeFlow(x_n, $x_e, X_L, \pm x_{action}$) are respectively the destination node, destination environment, a finite list of source (node, environment) pairs and a name for the one step flow. safeFlow($x_n, x_e, X_L, \pm x_{action}$) holds if the one step flow consisting of the source list X_L and the destination (x_n, x_e) named x_{action} is either permitted or prohibited depending upon the sign (+ or -) appearing in front of x_{action}.

Going back to the example in Figure 1, safeFlow(R_{temp}, [Alice,Cindy], [(y,[Alice,Cindy]), (z,[Alice, Bob,Cindy])], +binaryAdd) represents a one step flow that has R_{temp} as a destination node, [Alice,Cindy] as environment, [(y,[Alice,
Cindy]),(z,[Alice,Bob,Cindy])] as the source list consisting of (node,environment) pairs and `binaryAdd` as the action name. The (+) sign in front of the action sign (i.e. `+binaryAdd`) says that this one-step flow is permissible.

Stratum 2: Consists of a ternary predicate safeFlow*. The formal parameters x_{flowT}, x_{flowE} and x_{action} in safeFlow*($x_{flowT}, x_{flowE}, \pm x_{action}$) are respectively a flow tree, its environment x_{flowE} and its name x_{action}. safeFlow*($x_{flowT}, x_{flowE}, \pm x_{action}$), represents a permitted or prohibited flow tree, depending upon the sign (+ or -) appearing in front of x_{action}.

Going back to the example in Figure 1, safeFlow*([(R_{final}, e_{Rf}),[(z, e_z),[(R_{temp}, e_{Rt}), [(x, e_x),(y, e_y)]]]],[Alice, Bob,Cindy], +add) says that the tree rooted at R_{final} in the figure is permitted. Notice that the flow tree variable x_{flowT} has been instantiated to the list [(R_{final}, e_{Rf}),[(z, e_z),[(R_{temp}, e_{Rt}), [(x, e_x),(y, e_y)]]]]. The environment variable e_x, instantiated to the list [Alice,Bob,Cindy] is the union of all subjects that appeared in the access control lists of sources `x`, `y` and `z`. *add* is a name given to the tree that computes and stores the value `x+(y+z)`

Stratum	Predicate	Rules defining predicate
0	application-specific predicates List manipulation predicates	base application-specific relations. recursive list processing rules.
1	safeFlow	Body may contain literals from $\mathbf{SR}_0$.
2	safeFlow$*$	Body may contain safeFlow*and other literals from $\mathbf{SR}_0$ and $\mathbf{SR}_1$. Occurrences of safeFlow$*$ literal must be positive.
3	finalSafeFlow	The head is of the form finalSafeFlow(flowT,_,+_), the body may contain safeFlow, safeFlow*, and literals from $\mathbf{SR}_i$ for $i \leq 2$.
4	finalSafeFlow	The head of the form safeFlow*(flowT,_,−_), the body contains just one literal ¬safeFlow*(flowT,_,+_).

Table 1: Stratification of FlexFlow and Predicates

in the location R_{final}. e_{Rf}, e_{Rt}, e_x, e_y and e_z respective represents the environments of R_{final}, R_{temp}, x, y and z.

Strata 3 and 4: Consist of a ternary predicate finalSafeFlow, with the same arguments as safeFlow*, representing the flow control decisions finally made by FlexFlow. It is used to express conflict resolution policies. finalSafeFlow(x_{flowT},x_{flowE}, +x_{action}) holds when FlexFlow permits the flow tree x_{flowT} and is included in strata 3. finalSafeFlow(x_{flowT}, x_{flowE},-x_{action}) holds when FlexFlow prohibits the flow tree x_{flowT} and is included in strata 4.

Going back to the example in Figure 1, finalSafeFlow([(R_{final}, e_{Rf}), [(z, e_z), [(R_{temp}, e_{Rt}), [(x, e_x), (y, y_e)]]]], [Alice], +Add) permits the computation (x+y)+z). But finalSafeFlow([(R_{final}, e_{Rf}), [(z, e_z),(x, e_x),(y, y_e)]], [Alice],-Add) prohibits the computation x+y+z. Notice that this policy accepts left associative binary additions with a temporary location R_{temp} to store intermediate values, but not direct ternary addition of values stored in variables.

3.2 Rules used in FlexFlow

A FlexFlow specification consists of a finite set of Horn clauses (rules) constructed using above mentioned predicates. These rules are constructed so that they form a locally stratified logic program. Obtaining a locally stratified logic program requires that the predicates used in the rules belong to a finite set of *strata*, and the rules follow some syntactic constraints. Generally, predicates in the body of any Horn clause are from the lower strata than that of the head of the clause. The only exception to this rule occurs in stratum 2, where safeFlow* is allowed to appear in the head and the body of a rule. Rules that permit the head predicate to appear in the body have to satisfy further syntactic restrictions that they form a *local stratification*. Namely, the occurrence of safeFlow* in the body cannot be negative. Logically, this restriction implies that every instance of the recursive rule can be *unravelled* in a finite number of steps, where negation at any such *unravelling* is interpreted as failure over *previous unravelling*. FlexFlow is stratified by assigning levels to predicates as shown in Table 1, and the level of a rule is the level of its head predicate. Now we explain rules at each stratum with some examples.

Stratum 0: Rules in this stratum consists of basic facts related to application specific predicates and list processing functions. Following facts relate to the syntax tree example in Figure 1.

$$\begin{aligned} \mathsf{isEnv}(x, [Alice, Bob]) &\leftarrow \\ \mathsf{isEnv}(y, [Alice, Cindy]) &\leftarrow \\ \mathsf{isEnv}(z, [Alice, Bob, Cindy]) &\leftarrow \end{aligned}$$

These rules state that [Alice,Bob], [Alice,Cindy] and [Alice,Bob,Cindy] are the access control lists of variables x, y and z respectively.

Stratum 1: Rules is this stratum have literals from Stratum 0 in their bodies and heads that are instances of safeFlow. Example rules for the syntax tree example of Figure 1 is as follows.

$$\begin{aligned}\mathsf{safeFlow}(x_n, x_e, X_L, +binaryAdd) \quad \leftarrow \quad & \mathsf{union}([(u,u_e)],[(v,v_e)],X_L), \mathsf{isEnv}(u,u_e), \mathsf{isEnv}(v,v_e),\\ & \mathsf{intersection}(u_e,v_e,x_e)\end{aligned}$$

The rule says that it is safe for the source list X_L to *binaryAdd* into the root (x_n, x_e) provided that X_L has two elements (u, u_e) and (v, v_e), and the environment of x_e is the intersection of access lists of the sources. This rule also uses the list processing predicates union(A,B,C), intersection(A,B,C) that hold when C is the union/intersection of lists A and B respectively.

Stratum 2: Rules in this stratum have literals from Strata 0 and 1 in their bodies and heads that are instances of safeFlow*. Example rules for the syntax tree of Figure 1 are as follows.

$$\begin{aligned}
\mathsf{safeFlow}*(x_t, x_e, +balTree) \quad \leftarrow \quad & \mathsf{safeFlow}(x_n, x_e, [(u,u_e),(v,v_e)], +binAdd)\\
& \mathsf{union}(u_e, v_e, x_e), \mathsf{append}((x_n,x_e),[(u,u_e),(v,v_e)],x_t)\\
\mathsf{safeFlow}*(x_t, x_e, +balTree) \quad \leftarrow \quad & \mathsf{safeFlow}*(x_1, x_{e_1}, +balTree), \mathsf{safeFlow}*(x_2, x_{e_2}, +balTree)\\
& \mathsf{mkBinTree}(x_1,x_2,x_t), \mathsf{union}(x_{e_1}, x_{e_2}, x_e)\\
\mathsf{safeFlow}*(x_t, x_e, +balTree) \quad \leftarrow \quad & \mathsf{safeFlow}*(x_1, x_{e_1}, +balTree), \mathsf{safeFlow}*(x_2, x_{e_2}, +balancedTree),\\
& \mathsf{mkBinTree}(x_1,x_2,x_t), \mathsf{union}(x_{e_1}, x_{e_2}, x_e), \neg\mathsf{equal}(x_1,x_2)\\
\mathsf{safeFlow}*(x_t, x_e, -balTree) \quad \leftarrow \quad & \mathsf{safeFlow}*(x_1, x_{e1}, +balTree), \mathsf{safeFlow}*(x_2, x_{e2}, +balancedTree),\\
& \mathsf{union}(x_1,x_2,x_t), \mathsf{union}(x_{e1}, x_{e2}, x_e), \mathsf{equal}(x_1,x_2)
\end{aligned}$$

The first rule says that a one step flow tree is a safe flow tree. The second rule constructs larger flow trees from smaller ones. The larger tree is made by using the predicate mkBinTree. mkBinTree(x_1, x_2, x_t) makes a binary tree rooted at x_t with first and second children x_1 and x_2 respectively. In addition, the environment of the new tree is the union of environments of the two trees used to make up the larger tree. The third rule permits the construction of balanced binary trees, but ensures the constraint that no two children are shared. The forth rule explicitly prohibits balanced binary trees created by sharing children. These rules also use the predicate equal, where equal(x_1, x_2) holds when x_1 and x_2 are the same node. Hence, the third and the fourth rules are restrained versions of the second rule.

Stratum 3: Rules in this stratum may contain literals from stratum 0 through 2 in their bodies but only finalSafeFlow heads that have (+) action terms. They are used to specify conflicts that are resolved in favor of permissions. Example rules for the syntax tree of Figure 1 is as follows.

$$\begin{aligned}
\mathsf{finalSafeFlow}(x_t, x_e, +balTree) \quad \leftarrow \quad & \mathsf{safeFlow}*(x_t, x_e, +balTree), \mathsf{isMember}(Alice, x_e), \mathsf{isMember}(Bob, x_e)\\
\mathsf{finalSafeFlow}(x_t, x_e, +balTree) \quad \leftarrow \quad & \mathsf{safeFlow}*(x_t, x_e, +balTree), \mathsf{isLeafAccessList}(x_t, X), \mathsf{equal}(X, x_e).
\end{aligned}$$

The first rule says that a flow tree is safe provided that Alice and Bob are included in its environment. The second rule uses an extra predicate isLeafAccessList. isLeafAccessList(x_t, X) holds iff X is the union of all access lists of the leaves of x_t. The second rule says that a flow tree is safe provided that the environment of the tree is exactly the union of access control lists of the leaves.

Stratum 4: This stratum has one rule only. It is inserted by the FlexFlow system automatically to ensure the completeness . It reads as follows.

$$\mathsf{finalSafeFlow}(x_t, x_e, -x_{action}) \quad \leftarrow \quad \neg\mathsf{finalSafeFlow}(x_t, x_e, +x_{action})$$

This rule says that permissions not derivable using given rules are prohibited by the system.

3.3 Semantics of FlexFlow

The semantics of FlexFlow is given through the well known stable model semantics [GL88] and well founded model semantics [Gel89] of logic programs. In fact, as we showed in Section 3.2, FlexFlow specifications are locally stratified. This property guarantees that their stable model semantics is equivalent to their well founded semantics, thus ensuring that they have exactly one stable model (this follows from a result of Baral and Subrahmanian [BS92]).

4 Using FlexFlow to Express Existing Flow Control Models

This section shows how FlexFlow can express existing flow control models and their flow control policies. The first is the *lattice based flow control model* proposed by Denning in [Den76]. The second is the *decentralized label model* proposed by Myers and Liskov in [ML97, ML98, ML00]. The third is the *flexible information flow control model* proposed by Ferrari et al. in [FSBJ97]. The choice of three three existing models are based on following considerations.

Difference in system levels: The first model is an information flow control model designed for control flows at an application level. The second model is applicable for detailed runtime-level data-flow analysis [Koz99]. The third model control data flows within database transactions, viewing a database transaction as a tree of method calls over an object oriented system.

Difference between mandatory vs. discretionary permissions: The first model uses mandatory permissions (MAC), where information flow control is based on the security labels assigned to objects. The second and the third models are based on discretionary permissions (DAC) where every data item has a read-access control list (RACL). Flow control is based on these RACLs.

Difference in the degree of flexibility: The first model enforces a rigid flow control policy. The second model relaxes this rigid flow control policy by allowing owners of an object to specify their own flow policies by using decentralized labels. The third model relaxes the rigid flow control policy by allowing trusted methods to add extra readers to RACLs of objects they manipulate.

4.1 The Lattice Based Flow Control Model of Denning [Den76]

In [Den76], an information flow model FM is defined as $< N, P, SC, \oplus, \rightarrow >$, where N is a set of *objects*, P is a set of *processes* and SC is a set of disjoint *security* classes. $\oplus$ is a binary operator on SC and $\rightarrow$ specifies permissible flows among security classes. That is, for security classes A and B, $A \rightarrow B$ iff information in class A is permitted to flow into class B. Under some assumptions, referred to as *Denning's Axioms*, $< SC, \rightarrow, \oplus >$ form a universally bounded lattice where $< SC, \rightarrow >$ forms a partially ordered set.

Suppose f is an n-ary computable function, and a_i are object belonging to security classes $\underline{a_i}$ for all $i \leq n$. Then the flow control policy enforced by FM is as follows. If a value $f(a_1, \ldots, a_n)$ flows to an object b that is statically bound to a security class $\underline{b}$, then $\underline{a_1} \oplus \cdots \oplus \underline{a_n} \rightarrow \underline{b}$ must hold. If $f(a_1, \ldots, a_n)$ flows to a dynamically bound object b, then (if need be) the class $\underline{b}$ must be updated so that $\underline{a_1} \oplus \ldots \underline{a_n} \rightarrow \underline{b}$ holds.

To specify this flow control policy, we use the following sets. C is a set of classes, Obj is a set of objects, P is a set of processes. In this example, we use *objects* as *nodes* and *classes* as *environments of nodes* and the root's environment as the tree's environment. In addition, we define application specific predicates shown in Table 2. For instance, $\mathsf{dominate}(c, c')$ says that the class c dominates class c' in the class lattice C. The FM policy is now formulated as follows.

Predicate	Argument Types	Intended Meaning
$\mathsf{dominate}(c, c')$	(class, class)	Class c dominate class c' in the class set C.
$\mathsf{class}(o, c)$	(object,class)	Object o belongs to class c.
$\mathsf{leastUB}(cList, c)$	(class list,class)	The least upper bound of the elements in the class list $cList$ is c.
$\mathsf{setClass}(X, Y)$	((object,class) pair list, class list)	The classes of objects in X constitute class list Y.

Table 2: Predicates used to express Lattice based Flow Control Model of Denning [Den76]

$$
\begin{aligned}
\mathsf{class}(o_1, c_1) &\leftarrow \\
\mathsf{class}(o_2, c_2) &\leftarrow \\
\mathsf{dominate}(c_1, c_2) &\leftarrow \\
\mathsf{dominate}(x_c, z_c) &\leftarrow \mathsf{dominate}(x_c, y_c), \mathsf{dominate}(y_c, z_c). \\
\mathsf{safeFlow}(x_o, x_c, X, +x_{action}) &\leftarrow \mathsf{class}(x_o, x_c), \mathsf{setClass}(X, Y), \mathsf{leastUB}(Y, x), \mathsf{dominate}(x_c, x). \\
\mathsf{safeFlow}*([(x_o, x_c) \mid X], x_c, +x_{action}) &\leftarrow \mathsf{safeFlow}(x_o, x_c, X, +x_{action}). \\
\mathsf{finalSafeFlow}(X, x_c, +x_{action}) &\leftarrow \mathsf{safeFlow}*(X, x_c + x_{action})
\end{aligned}
$$

The first two rules specify that o_1, o_2 are associated with c_1 and c_2 respectively. The third rule says that c_1 dominate c_2 in C. The fourth rule is the transitivity of the dominance relation. The fifth rule says that information from objects in the list X is allowed to flow into x_o iff the least upper bound of the classes of objects of X is dominated by the class of x_o. The sixth rule constructs the depth-one permissible flow tree $[(x_o, x_c) \mid X]$ with environment x_c. Because the flow control policy in this lattice based flow control model has only one-step flows, we don't need to construct flow trees of depth greater that one. Furthermore, the model doesn't allow negative permission, so the conflict resolution policy is simple and given by the last rule. Because FlexFlow does not have a facility to change security labels as a consequence of permitted flows (fixing that would require enforcing system obligations [BJWW02]) we do not model updating security classifications for dynamically bound objects.

4.2 The Decentralized Label Model of Mayer and Liskov [ML97, ML98, ML00]

The *Decentralized Label Model* [ML97, ML98, ML00] of Mayer and Liskov is applied at the runtime data flow analysis level. This model has variables storing values and updating them during a computation. Values are fed from external sources to the computation by means of special variables referred to as input channels, and values are fed back to external sinks through special variables referred to as output channels. Values and variables occurring in computations have labels. A label contains a list of owners whose data was observed in order to construct the data value in question. Each owner declares a set of principals that may read the value, referred to as the reader set of that owner. Each (owner, reader set) pair is said to constitute a per-principal flow control policy. When a value is read from a variable or an input channel, the value acquires the label of the variable or the input channel. When a value is written to a variable, a new copy of the value is generated with the label of the variable. The model also uses a principal hierarchy where some principals are authorized to act for others.

The flow control policy used in this model is that when information flows from one object (here, object refers variable or channel) to another, the label of the destination must be more restrictive than the label of the source. A label L_1 is said to be more restrictive than label L_2 iff L_1 contains all the owners of label L_2, and the same or fewer readers for each owner. In other word, a label L_1 is restrictive than label L_2 iff the effective reader set of L_1 is a subset of label L_2. The effective reader set of a label is the set of readers common to all owners in the label. In addition, [ML97] allows a principle to read an item provided that it can act for some other permitted readers. Furthermore, the policy allows owners to add readers, thereby allowing information declassification.

To express the policies of [ML97], we define Obj, P and L as the sets of variables, principals and labels respectively. We use variables (input and output channels included) as nodes of the flow trees and labels of variables

Predicate	Parameter Types	Intended Meaning
$\mathsf{label}(o, l)$	(object,label)	l is the label of the object o.
$\mathsf{eRdSet}(l, X)$	(label, principal list)	The effective reader set of label l is X.
$\mathsf{cover}(X, Y)$	(principal list, principal list)	Every principal in X can act for some principals in Y.
$\mathsf{listOfRdSet}(X, Y)$	((obj,label) pair list,list of RdSet)	The reader sets of all the labels in X constitute the list Y.
$\mathsf{allIntersec}(X, Y)$	(list of principal list, principal)	The intersection of all the principal lists in X is Y.

Table 3: Predicates used to express Decentralized Label Model of Myers and Liskov [ML97]

.

as environments of the nodes. Before specifying flow control policies as rules, we define some application specific predicates as shown in Table 3. For example, the predicate $\mathsf{label}(o, l)$ holds iff l is the label of o and $\mathsf{eRdSet}(l, X)$ hold iff the effective reader set of label l is X. Flow control policies of [ML97] are now specified in FlexFlow using the following rules.

$$
\begin{aligned}
\mathsf{safeFlow}(x_o, x_l, Y, +x_{action}) &\leftarrow \mathsf{label}(x_o, x_l), \mathsf{listOfRdSet}(Y, Y'), \mathsf{allIntersec}(Y', W), \\
&\quad\ \mathsf{eRdSet}(x_l, X), \mathsf{cover}(X, W) \\
\mathsf{safeFlow}*([(x_o, x_l) \mid Y], x_l, +x_{action}) &\leftarrow \mathsf{safeFlow}(x_o, x_l, Y, +x_{action}) \\
\mathsf{finalSafeFlow}(X, x_l, +x_{action}) &\leftarrow \mathsf{safeFlow}*([(x_o, x_l) \mid Y], x_l, +x_{action})
\end{aligned}
$$

The first rule says that information can flow from objects in the list Y to x_o provided that each of principals in the effective reader set of label x_l can act for some principals in the intersections of the effective reader sets of labels in Y. The second rule constructs the depth one permissible flow tree. The third rule specifies the conflict resolution policy.

4.3 The Flexible Information Flow Control Model of Ferrari et al. [FSBJ97]

This model controls the flow of information within transactions [FSBJ97]. Here, a transaction is decomposed into a tree of method calls. Methods manipulate data objects, and invoke other methods on objects. Based on the invocation relations between the executions, a transaction is represented as a *transaction execution tree*, where the nodes are executions of methods and branches are caller-calee relations between methods. The root of the tree is the execution invoked by the initiator of the transaction. For example, if execution e_j invokes e_h, then e_h is a child of e_j in the transaction execution tree. Each execution has a method name `m` and an object `o` on which it executes symbolized by the pair `(m,o)`. Figure 3 gives an example *transaction execution tree* simplified from the example in [FSBJ97]. As used in [FSBJ97], we use $(m, \_)$ when the object that is manipulated by m is not relevant for the discussion.

Forward and backward channels for information flow are created from the transaction execution tree. Given executions e_i, e_j and e_k, there is said to be a *forward channel* from e_i to e_j if e_i invokes e_k, or e_i invokes e_j and there is a forward channel from e_j to e_k. There is a *backward channel* from e_k to e_i if e_i invokes e_k or e_i invokes e_j and there is a backward channel from e_k to e_j. There is an *information flow* from o_h to o_k by e_h and e_k, written (o_h, e_h, e_k, o_k) if the method of e_h is *read*, the method of e_k is *write*, and there is a transmission channel (perhaps a combination of some forward and backward channels). According to these definitions, there are four information flow trees derivable from the transaction execution tree of Figure 3. They are (o_3, e_3, e_{17}, o_6), (o_4, e_9, e_{17}, o_6), $(o_5, e_{13}, e_{17}, o_6)$, and $(o_5, e_{16}, e_{17}, o_6)$.

Every object has an access control list (ACL) specifying the users allowed to read and write the object. Based on these ACL's, two flow control policies can be specified. In the *strict policy*, a flow (o_h, e_h, e_k, o_k) is safe iff the ACL of o_k is a subset of the ACL of o_h. The *flexible policy* of [FSBJ97] makes exceptions to the strict policy by allowing trusted methods to retrieve and transmit information without obeying the *strict policy*. In order to do so, each method has *reply* and *invoke* waivers associated with it. Both waivers consist of a set of pairs (O, U), where O is a set of objects, U is a set of users. The flexible policy works as follows. Suppose `m` is a trusted method with a reply waiver $RW(m)$ and an invoke waiver $IW(m)$. If $(O_i, U_i) \in RW(m)$, then the information in O_i can be released to users in U_i through m's reply even if users in U_i are not authorized to read object o_i. If $(O_i, U_i) \in IW(m)$, m is allowed

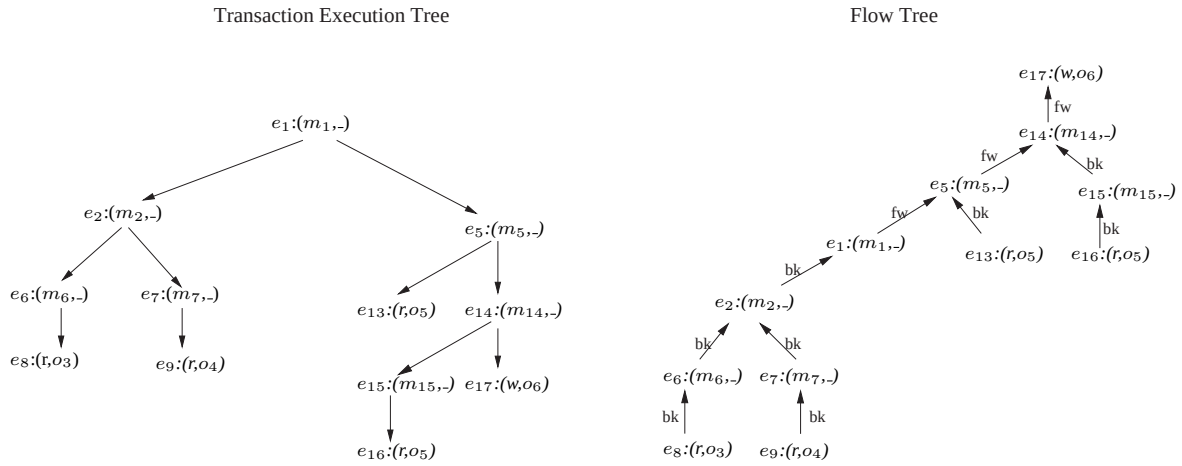

Figure 3: A Transaction Execution Tree and its Flow Tree

to write information read from O_i to other objects that the users in U_i can read even if these users are not allowed to access O_i. Thus, under the flexible policy, a flow (o_h, e_h, e_k, o_k) is safe iff the union of the reader list and the waiver lists for o_h along the transmission channel from o_h up to but excluding o_k subsume the readers list of o_k. The execution of a write method is said to be safe iff all flows ending at it are safe.

Because information flow policy applies to flow trees that are embedded in transaction execution trees, we provide an algorithm referred to as the *flow tree generator* to extract all flow trees of a transaction execution tree in Appendix A. We do so because flow policies apply to flow trees and not directly to transaction execution trees. In flow trees, the roots have write methods and leaves have read methods. Edges are labelled either forward (`fw`) or backward (`bw`). Figure 3 shows a flow tree embedded in the example transaction execution tree.

Based on the flow trees extracted from the transaction execution tree, we show how to express the flexible flow control model of [FSBJ97] using FlexFlow, and four sets, Obj, U, M, E for objects, users, methods and executions respectively. We represent a waiver as a list of $[o_i, U_i]$ pairs where o_i is an object, U_i is a set of users who are allowed to access the information in o_i because of the waiver. For example, $[[o_1, Frank, David], [o_3, Frank]]$ is a waiver implying that $Frank$ and $David$ can access o_1 and $Frank$ can access o_3 because of this waiver. Our nodes are executions, and a flow tree environment has all waivers collected along all branches that go up to the root. In this example we use predicates to associate methods and objects with nodes and executions. In this example, we do not need node environments, and accordingly they are empty lists. Our predicates are given in Table 4. Some of these are not standard list manipulation predicates, and therefore we show their definitions in Appendix A.1. Using these predicates some sample rules that fit in stratum 0 of FlexFlew are as follows.

$$\begin{aligned}
\mathsf{mtd}(e_8, r) &\leftarrow \\
\mathsf{obj}(e_8, o_3) &\leftarrow \\
\mathsf{rAcl}(o_3, [Ann, Bob, Carol, David]) &\leftarrow \\
\mathsf{rW}(m_3, [[o_1, Frank], [o_3, Frank]]) &\leftarrow \\
\mathsf{iW}(m_3, [[o_4, David], [o_5, David]]) &\leftarrow \\
\mathsf{fwFlowCH}(e_{14}, e_5) &\leftarrow
\end{aligned}$$

The first two rule specify that e_8 is a read method of o_3. The third rule says that [Ann,Bob,Carol,David] is the access control list of o_3. The fourth and fifth rules say that m_3's reply and invoke waivers are $[[o_1, Frank], [o_3, Frank]]$ and $[[o_4, David], [o_5, David]]$ respectively. The last rule says that there is a one-step forward flow channel from e_5 to e_{14}. One-step information flows are specified using the following two rules.

$$\begin{aligned}
\mathsf{safeFlow}(x_e, v, [(y_e, v)], +bkChannel) &\leftarrow \mathsf{bkFlowCH}(x_e, y_e). \\
\mathsf{safeFlow}(x_e, v, [(y_e, v)], +fwChannel) &\leftarrow \mathsf{fwFlowCH}(x_e, y_e).
\end{aligned}$$

Predicate	Attribute Types	Intended Meaning
obj(e, o)	(exec., object)	Execution e is executed on object o.
mtd(e, m)	(exec., method)	The method of execution e is method m.
rAcl(o, X)	(object, list of users)	The access control list of object o is X.
rW(m, X)	(method,waiver list)	X is the reply waiver list associated with method m.
iW(m, X)	(method,waiver list)	X is the invoke waiver list associated with method m.
releaseBK(X, m, X')	(flowE, method, flowE)	The result of applying reply waiver of method m to flow tree environment X is X'.
releaseFW(X, m, X')	(flowE, method, flowE)	The result of applying invoke waiver of method m to flow tree environment X is X'.
unionAdd(X, Z)	(list of flowE, flowE)	The union-add of flow environments in X is Z.
eRdSet(X, Y)	(flowE, user list)	The effective readers set of flow tree environment is Y.
subset(X, Y)	(user list, user list)	User list X is subset of user list Y.
bkFlowCH(e_i, e_j)	(exec., exec.)	There is a backward edge from e_j to e_i.
fwFlowCH(e_i, e_j)	(exec., exec.)	There is a forward edge from e_j to e_i.
listOfSafeFlow$((x, e), Y, a)$	((exec.,env.),list, action)	For all $y_i \in Y$, safeFlow$((x, e), y_i, +a)$ holds.
listOfSafeFlow*(X, Y, E, a)	(list, list, list,action)	For all $(x_i, e_i) \in X$, $y_i \in Y$, and $e_i \in E$, safeFlow*$(y_i, e_i, +a)$ and isHead$((x_i, e_i), y_i)$ hold.
listOfMtd(X, M)	(list, list)	For all $x_i \in X$, $m_i \in M$, mtd(x_i, m_i) holds.
listOfReleaseBK(X, M, Y)	(list, list, list)	For all $x_i \in X$, $m_i \in M$, and $y_i \in Y$, releaseBK(x_i, m_i, y_i) holds.

Table 4: Predicates used to express the Flexible Flow Control Model of Ferrari et al. [FSBJ97]

These two rules say that in order to have one-step forward/backward flow, there must be a one-step forward/backward channel from the source to the sink. Permissible information flow trees are constructed using the following recursive rules.

$$
\begin{aligned}
\mathsf{safeFlow}*([x_e],[x_o|X],+u) \quad &\leftarrow \quad \mathsf{obj}(x_e, x_o), \mathsf{mtd}(x_e, r), \mathsf{rAcl}(x_o, X) \\
\mathsf{safeFlow}*(X_{flowT}, X_{flowE}, +n) \quad &\leftarrow \quad \mathsf{mtd}(x_e, x_m), \mathsf{listOfSafeFlow}((x_e, v), Y, +bkChannel), \\
&\qquad \neg\mathsf{mtd}(x_m, w), \mathsf{listOfSafeFlow}*(Y, Y_{flowT}, Y_{flowE}), \\
&\qquad \mathsf{listOfMtd}(Y, Y_m), \mathsf{listOfReleaseBK}(Y_{flowE}, Y_m, Y'_{flowE}), \\
&\qquad \mathsf{unionAdd}(Y'_{flowE}, X_{flowE}), \mathsf{append}((x_e, v), Y_{flowT}, X_{flowT}) \\
\mathsf{safeFlow}*(X_{flowT}, X_{flowE}, +n) \quad &\leftarrow \quad \mathsf{safeFlow}(x_e, v, [(z_e, v)], +fwChannel), \mathsf{mtd}(z_e, z_m), \\
&\qquad \mathsf{safeFlow}*(Z_{flowT}, Z_{flowE}, +n), \mathsf{isHead}((z_e, v), Z_{flowT}), \\
&\qquad \mathsf{listOfSafeFlow}((x_e, v), Y, +bkChannel), \\
&\qquad \mathsf{listOfSafeFlow}*(Y, Y_{flowT}, Y_{flowE}), \\
&\qquad \mathsf{listOfMtd}(Y, Y_m), \mathsf{listOfReleaseBK}(Y_{flowE}, Y_m, Y'_{flowE}), \\
&\qquad \mathsf{releaseFW}(Z_{flowE}, z_m, Z'_{flowE}), \\
&\qquad \mathsf{append}(Y'_{flowE}, z'_{flowE}, X'_{flowE}), \\
&\qquad \mathsf{unionAdd}(X'_{flowE}, X_{flowE}), \mathsf{append}((x_e, v), Z'_{flowT}, X'_{flowT}), \\
&\qquad \mathsf{append}(X'_{flowT}, Y_{flowT}, X_{flowT})
\end{aligned}
$$

$$\begin{aligned}
\mathsf{safeFlow}*([x_e,[y_e|Tail],X,+n) &\leftarrow \mathsf{safeFlow}(x_e,v,[(y_e,v)],+fwChannel),\mathsf{mtd}(x_e,w),\\
&\quad \mathsf{safeFlow}*([y_e|Tail],Y,+n),\mathsf{mtd}(y_e,y_m),\\
&\quad \mathsf{releaseFW}(Y,y_m,Y'),\mathsf{eRdSet}(Y',Y''),\mathsf{obj}(x_e,x_o),\\
&\quad \mathsf{rAcl}(x_o,X),\mathsf{subset}(X,Y'')\\
\mathsf{finalSafeFlow}(X_{flowT},X_{flowE},+x_{action}) &\leftarrow \mathsf{safeFlow}*(X_{flowT},X_{flowE},+x_{action}).
\end{aligned}$$

The first rule says that executing a read method m constructs a one node flow tree that has $[o,X]$ as its environment where X is the ACL of o. Second and third rules recursively constructs flow trees that have non-write roots. Note that for any node in any flow tree extracted from a given transaction execution tree, there is at most one forward channel feeding into it. The fourth rule constructs the whole permissible flow tree whose root is a write execution. The last rule resolves conflicts.

4.4 Comparison

As seen from Sections 4.1, 4.2, and 4.3, FlexFlow can be used to express flow control policies presented in some prior models. The first two models we used are explicitly for one-step flows. That is their flows are between a collection of sources and a single destination, and therefore their flow control policies govern one-step flows. Flows in 4.3 are from multiple sources to a multiples destinations, but flow policies apply to branches that connects each source to its final destination. Because FlexFlow maintains flow trees, it is capable of modelling both kinds of flows. But in addition, FlexFlow is also capable of specifying policies that can combine multiple paths at any intermediate node, and thereby going a step beyond these models.

In addition, we show how to automatically extract flow trees from a transaction tree given in Section 4.3. Therefore, it is possible to automatically check for policies governing flow control in such transaction systems using an implementation of FlexFlow.

Additionally, existing models such as those we have shown in Sections 4.1, 4.2, and 4.3 only allow permissions. But FlexFlow also allows prohibitions. In addition, as will be shown shortly, FlexFlow can specify some constraints and resolve them.

5 Specifying Constraints

This section shows how to specify static flow constraints using FlexFlow. Enforcing dynamic flow constraints requires formalizing the execution of FlexFlow trees, a direction currently under study.

In order to motivate the use of static flow constraints, consider a file systems that allows users to copy each other's files subjected to some copy protection policies. To make the example concrete, let the file system have a set of objects, users and their roles given as Obj,U and R respectively. The roles form a role hierarchy, referred as RH is shown in Figure 4. For simplicity, we assume that each user plays a single role and each object has a unique owner. The policy governing the copy protection read as follows. Different source files can be copied into a single destination file iff the roles of the owners of the source files are dominated by the role of the owner of the destination file.

In order to show how to specify this copy protection policy, we use objects as nodes and owners as node environments. We also use some auxiliary predicates summarized in Table 5. For example, $\mathsf{userList}(X_l,X_u)$ says that user list of the source (object, user) pair list X_l is X_u. $\mathsf{roleList}(X_u,X_r)$ says that the corresponding roles played by a list of users X_u is X_r. $\mathsf{listDominate}(x_r,X_r)$ says that role x_r dominates each role in the list of roles X_r. These three auxiliary predicates can be defined using normal Prolog rules. For example, we show the definition for $\mathsf{listDominate}(x_r,X_r)$.

$$\begin{aligned}
\mathsf{listDominate}(x,[\,]) &\leftarrow\\
\mathsf{listDominate}(x,X) &\leftarrow \mathsf{isHead}(y,X),\mathsf{isTail}(Y,X),\mathsf{listDominate}(x,Y),\mathsf{in}(y,x)
\end{aligned}$$

Using the predicates we state the copy protection policy as follows.

Predicate	Argument Types	Intended Meaning
$\mathsf{owner}(o, u)$	(object,user)	User u is the owner of o.
$\mathsf{role}(u, r)$	(user,role)	User u plays role r.
$\mathsf{in}(r_1, r_2)$	(role, role)	Role r_1 is dominated by role r_2 in role hierarchy RH.
$\mathsf{userList}(X, Y)$	((o, u) pair list, user list)	The user list of the source (object, user) pair list X is Y.
$\mathsf{roleList}(X, Y)$	((o, u) pair list, role list)	The corresponding roles played by a list of users X is Y.
$\mathsf{listDominate}(x, Y)$	(role, role list)	The role x dominates each role in the role list Y.

Table 5: Predicates used in the Copy Protection Policy

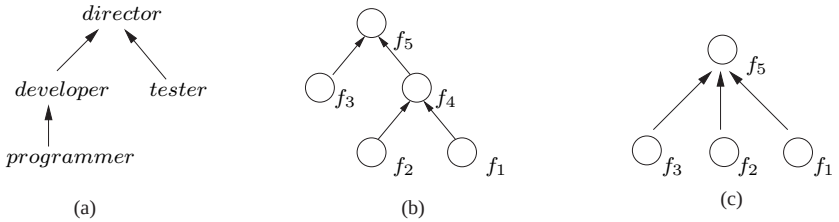

Figure 4: A Role Hierarchy and Flow Trees that Respect Role Constraints

$$
\begin{aligned}
\mathsf{owner}(f_1, Alice) &\leftarrow \\
\mathsf{owner}(f_2, Bob) &\leftarrow \\
\mathsf{owner}(f_3, Cindy) &\leftarrow \\
\mathsf{owner}(f_4, David) &\leftarrow \\
\mathsf{owner}(f_5, Eric) &\leftarrow \\
\mathsf{role}(Alice, programmer) &\leftarrow \\
\mathsf{role}(Bob, programmer) &\leftarrow \\
\mathsf{role}(Cindy, tester) &\leftarrow \\
\mathsf{role}(David, developer) &\leftarrow \\
\mathsf{role}(Eric, director) &\leftarrow \\
\mathsf{safeFlow}(x_o, x_u, X_l, +copy) &\leftarrow \mathsf{role}(x_u, x_r), \mathsf{userList}(X_l, X_u), \mathsf{roleList}(X_u, X_r), \\
&\quad \mathsf{listDominate}(x_r, X_r) \\
\mathsf{safeFlow}*(X_{flowT}, [\,], +copy) &\leftarrow \mathsf{safeFlow}(x_o, x_u, X_l, +copy), \mathsf{isHead}((x_o, x_u), x_{flowT}), \\
&\quad \mathsf{isTail}(X_l, x_{flowT}) \\
\mathsf{safeFlow}*(X_{flowT}, [\,], +copy) &\leftarrow \mathsf{safeFlow}(x_o, x_u, X_l, +copy), \mathsf{listOfSafeFlow}*(X_l, Y_{flowT}), \\
&\quad \mathsf{append}((x_o, x_u), Y_{flowT}, X_{flowT})
\end{aligned}
$$

The first ten rules state basic facts about file ownership and user roles. The 11th rule specifies safe one step flows enforcing the stated copy protection policy. The 12th rule says that a one step flow constitutes a safe depth-one flow tree. The last rule constructs larger flow trees from the smaller ones. Notice that the last rule uses an auxiliary predicate $\mathsf{listOfSafeFlow}*(X_l, Y_{flowT})$, that is definable as follows. Predicates append, isHead, and isTail used in the definition of listOfSafeFlow* are standard list manipulation predicates.

$$
\begin{aligned}
\mathsf{listOfSafeFlow}*([\,], [\,]) &\leftarrow \\
\mathsf{listOfSafeFlow}*([x|X], [Y|Z]) &\leftarrow \mathsf{safeFlow}*(Y, [\,], +copy), \mathsf{isHead}(x, Y), \mathsf{listOfSafeFlow}*(X, Z)
\end{aligned}
$$

Based on stated rules and the role hierarchy shown in Figure 4, many permissible flow trees are derivable. Two such example are $[(f_5, Eric), (f_3, Cindy), [(f_4, David), (f_1, Alice), (f_2, Bob)]]$ and $[(f_5, Eric), (f_3, Cindy),$

$(f_1, Alice)$, (f_2, Bob)], as shown in Figure 4(b) and (c). Now suppose that *programmer* and *tester* are conflicting roles and we want to enforce the policy that roles of source objects owners do not conflict. This additional requirements can be enforced in FlexFlow by changing the 11th rule to read as follows.

$$\mathsf{safeFlow}(x_o, x_u, X_l, +copy) \leftarrow \mathsf{role}(x_u, x_r), \mathsf{userList}(X_l, X_u), \mathsf{roleList}(X_u, X_r), \mathsf{listDominate}(x_r, X_r), \neg\mathsf{conflictList}(X_r)$$

Here conflictList(X_r) holds iff the role list X_r contains at least one pair of conflicting roles. This predicate can be defined using the following rule, where $conflict(x, y)$ holds iff roles x and y conflict. isMember(x, X) holds iff x is an element of X.

$$\mathsf{conflictList}(X) \leftarrow \mathsf{isMember}(x, X), \mathsf{isMember}(y, X), \mathsf{conflict}(x, y)$$

Using the new rule for safeFlow instead of the original one, some flow trees that were acceptable under the old policy now become unacceptable. In the given example, the flow tree of Figure 4(c) is no longer permissible, but the flow tree Figure 4(b) is still permissible. That says information from file f_1, f_2 and f_3 can not be copied into file f_5 in one step. But they can be copied into file f_5 in two steps. That is, to copy information from f_1 and f_2 into file f_4 in first step and then to copy information from f_3 and f_4 into f_5 in the second step.

5.1 Prologue

Notice that the given constraint resolution rules enforce a Chinese wall policy. In terms of flow trees, they resolve static inter-node conflicts. Thus, FlexFlow constructs those tree that do not violate stated constraints because it is able to exploit the list construction capabilities in logic programming. Notice that frameworks such as FAF specifies specifies static constraints using Horn clauses that have a `false` (i.e. unsatisfiable) head, but do not always resolve them.

Our example do not explain two other kind of constraints that can be specified and resolved by the same technique. The first kind is intra-tree conflicts. In order to see the difference, notice that constraints that were avoided in the example policy are those that existed between nodes, i.e. intra-tree conflicts within a single tree. In addition we can specify and enforce constraints that are between two trees, i.e. inter-tree conflicts. For example, a policy could require that David could only be the source of two copy trees. The resolution of such a policy requires FlexFlow to keep an accurate account of flow trees that have been constructed, and to avoid having two trees with David's files in their leaves. This kind of conflict resolution uses lists of basic objects of study (trees in our case) and predicates about them. We have proposed that for access control elsewhere [WJ03].

The second kind of constraints are dynamic constraints that limits ways in which flow trees can be executed. It is possible that two trees can execute in isolation, but if a particular flow had occurred in flow tree (such as Frank has already copied David's file in one tree) then another flow could not occur in a different flow tree (such as now Mary cannot copy any of Franks files in another tree). Specifying and enforcing such constraints requires the formalization of flow executions in flow trees, and that form some of our ongoing work.

6 Related Work

Flow control is a heavily researched area in the context of multi-level security. In the early days, Bell and La-Padula [BL75] and Denning [Den76] proposed the lattice based models of information flow. There, objects and subjects have security labels. The labels form a partially ordered lattice. Information flow control prevents information flowing from higher levels to lower levels. Some flexibility to Denning's lattice based model was later added by Foley [Fol89], where each entity was associated with a label pair instead of a single label.

McCollum et al. [MMN90] presented an Owner Retained Access Control model (ORAC) to control information dissemination. In ORAC, the user who creates a data object is considered its owner and has the right to define an ACL for it. The object's owner and its ACL is carried in the object label and used for access control. With information

flowing from object to object, an object may have multiple owner and corresponding ACLs. Here, dissemination control is achieved by ensuring that all readers of an object are in all the ACLs associated with it.

Samarati et al. [SBCJ97] presents an information flow control model for object-oriented system. In that model, a process can write an object o only if o is protected in reading *at least as* all objects read by the process up to that point. An object o is *at least as* protected as another object o' if the read ACL of o is contained in the read ACL of o'. The information flow control policy in the model in is the *strict* policy [SBCJ97].

Bertino et al. [BA99] presented a language to express both static and dynamic authorization constraints as clauses in a logic program. Although we do not show how to model workflow as elaborately as Bertino et al., we show how to model security properties such as static separation of duty in this paper. In addition, we can model graphs with splits and joins that appear in the work of Bertino et al. [BA99].

Our work has been influenced by the *Flexible Authorization Framework* (FAF) of Jajodia et al. [JSSS01]. FAF is specifies access control policies, but not flow control policies. As in FlexFlow, FAF is also based on Horn clauses with stable model semantics. Thus FlexFlow take the same paradigm to flow control, but does so with lists of objects, instead of objects themselves. In addition, work described in Section 4 as well as those mentioned in this section have flow control frameworks tailored for specific application domains. But FlexFlow is domain independent, and meta-policy independent.

7 Conclusions

The FlexFlow framework specifies flow control policies as stratified logic programs consisting of five levels. The first level allows the specifier to build his/her own structures to express the flow control policies. The second and the third levels are used to propagate flows. Because FlexFlow allows the derivation of permissions and prohibitions on flows, there is a need to specify *conflict resolution* aspect of flow control policies. The fourth level does that. The fifth level completes the polices, in the sense that it enforces that every flow request is either accepted or rejected.

By expressing flow control policies at the programming language and transaction specification we have shown the generality of FlexFlow. Thereby it can be used to compare and combine flow control polices at different levels. Currently, we are unaware of such a framework.

Ongoing work on FlexFlow includes designing materialization structures, adding revocations and constructing constraint compliant flows. Another important issue we are working on is the specification and enforcement of dynamic constraints on flows.

References

[BA99] E. Bertino and V. Atluri. The specification and enforcement of authorization constraints in workflow management. *ACM Transactions on Information Systems Security*, 1(2):65–104, February 1999.

[BJWW02] C. Bettini, S. Jajodia, X.S. Wang, and D. Wijesekera. Obligation monitoring in policy management. In *Proc. 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 502–513, Monterey, CA, June 2002.

[BL75] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations and model. Report M74-244, Mitre Corp., Bedford, MA, 1975.

[BS92] C. Baral and V.S. Subrahmanian. Stable and extension class theory for logic programs and default theories. *Journal of Automated Reasoning*, 8:345–366, 1992.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 3 edition, 1987.

[Den76] D.E. Denning. A lattice model of secure information flow. *Communication of ACM*, pages 236–243, May 1976.

[Fol89] S.N. Foley. A model for secure information flow. In *Proceedings of the IEEE symposium on Security and Privacy*, Oakland, CA, May 1989.

[FSBJ97] E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 130–140, Oakland, CA, May 1997. IEEE.

[Gel89] A.V. Gelder. The alternating fixpoint of logic programs with negation. In *Proc. 8th ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.

[JSSS01] S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(4):1–57, June 2001.

[Koz99] D. Kozen. Language-based security. In *Proceedings of Mathematical Foundations of Computer Science*, pages 284–298. Springer-Verlag, 1999.

[ML97] A.C. Myers and B. Liskov. A decentralized model for information flow control. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 129–142, Saint-Malo, France, October 1997.

[ML98] A.C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the IEEE symposium on Security and Privacy*, Oakland, CA, May 1998. IEEE.

[ML00] A.C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ATM Transactions on Software Engineering and Methodology*, (4):410–442, 2000.

[MMN90] C.J. McCollum, J.R. Messing, and L. Notargiacomo. Beyond the pale of mac and dac-defining new forms of access control. In *Proceedings of the IEEE symposium on Security and Privacy*, pages 190–200, Oakland, CA, May 1990.

[SBCJ97] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia. Information flow control in object-oriented systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):524–538, July-Aug. 1997.

[WJ03] D. Wijesekera and S. Jajodia. Obtaining constraint compliant authorization lists in the flexible authorization framework. 2003. Submitted for publication.

A The Flow Tree Generator

Algorithm *Flow Tree Generator*
INPUT: Transaction execution tree T.
OUTPUT: A set of Flow trees: $flowT_1, \ldots, flowT_n$, n is the number of executions with "w" methods.
1.Let $flowTreeRoot =: \{e_r | e_r \in E, mtd(e_r, w)\}$
2.For each element of $flowTreeRoot$: e_r:

If $invoke(e_h, e_r)$: Insert e_h as child of e_r, Insert "$e_h \xrightarrow{fw} e_r$", $getchildren(e_h)$
end-for

Procedure $getchildren(current\ execution : e_j)$
If $\neg mtd(e_j, r)$:
a. If $invoke(e_i, e_j)$ & $precede(e_i, parent(e_j))$:
Insert e_i as a child of e_j, Insert arch "$e_i \xrightarrow{fw} e_j$", $getchildren(e_i)$
b.If $invoke(e_j, e_k)$ and e_k is not the parent of e_j:
If $\neg mtd(e_k, w)$: Insert e_k as a child of e_j, Insert arch "$e_k \xrightarrow{bk} e_j$", $getchildren(e_k)$
If $mtd(e_k, w)$: $removeparent(e_j)$
end-if
end-if

Procedure $removeparent(current\ execution : e_j)$
If e_j has no child:
delete e_j from his parent's children set, delete "$e_j \xrightarrow{bk} parent(e_j)$", $removeparent(parent(e_j))$
end-if

Table 6: The Flow Tree Generator

A.1 Some predicates used in Section 4.3

In this section, we define predicates that are not standard in Prolog, but used in section 4.3. We do so using standard list manipulation predicates given in Table 7, and defined in [CM87].

Predicate & Arity	Meaning
isHead(X, Y)	X is the head of list Y.
isTail(X, Y)	List X is the tail of list Y.
member(X, Y)	X is an element of Y.
union(X, Y, Z)	The union of list X and Y is list Z.
intersection(X, Y, Z)	The intersection of list X and Y is list Z.
append(X, Y, Z)	Join X and Y to get Z.
remove(X, Y, Z)	Remove Y from X to get Z.

Table 7: Some Standard List manipulation Predicates in [CM87]

Rules for unionAdd**(X,Z)**

$$\begin{aligned}
\mathsf{unionAdd}([X], X) &\leftarrow .\\
\mathsf{unionAdd}([Head|Tail], Z) &\leftarrow \mathsf{unionAdd}([Tail], Z'), \mathsf{binUnionAdd}(Head, Z', Z).
\end{aligned}$$

$$\begin{aligned}
\mathsf{binUnionAdd}(X, [\,], X) &\leftarrow .\\
\mathsf{binUnionAdd}([Head_1|X], [Head_2|Y], Z] &\leftarrow \mathsf{binUnionAdd}(X, Y, Z'), \mathsf{binBinUnionAdd}(Head_2, Z', Z''),\\
&\quad \mathsf{binBinUnionAdd}(Head_1, Z'', Z).
\end{aligned}$$

$$\begin{aligned}
\mathsf{binBinUnionAdd}(X, [\,], X) &\leftarrow .\\
\mathsf{binBinUnionAdd}(X, Y, Z) &\leftarrow \mathsf{isHead}(o, X), \mathtt{objSet}(O, Y), \neg\mathsf{member}(o, O), \mathsf{append}(Y, X, Z).\\
\mathsf{binBinUnionAdd}(X, Y, Z) &\leftarrow \mathsf{isHead}(o, X), \mathsf{member}(Y', Y), \mathsf{isHead}(o, Y'),\\
&\quad \mathsf{isTail}(Tail_1, Y'), \mathsf{isTail}(Tail_2, X),\\
&\quad \mathsf{intersection}(Tail_1, Tail_2, Tail), \mathsf{remove}(Y, Y', Y''),\\
&\quad \mathsf{append}(Y'', [o|Tail], Z).
\end{aligned}$$

Rules for releaseBK**(flowE, m, flowE')**

$$\begin{aligned}
\mathsf{releaseBK}(X, m, X') &\leftarrow \mathsf{rW}(m, RW), \mathsf{replyAdd}(RW, X, X').
\end{aligned}$$

$$\begin{aligned}
\mathsf{replyAdd}([\,], Y, Y) &\leftarrow .\\
\mathsf{replyAdd}([Head|Tail], Y, Y') &\leftarrow \mathsf{replyAdd}(Tail, Y, Y''), \mathsf{binReplyAdd}(Head, Y'', Y').
\end{aligned}$$

$$\begin{aligned}
\mathsf{binReplyAdd}([\,], Y, Y) &\leftarrow .\\
\mathsf{binReplyAdd}(X, Y, Z) &\leftarrow \mathsf{isHead}(o, X), \mathsf{member}(Y', Y), \mathsf{isHead}(o, Y'), \mathsf{isTail}(Tail_1, X),\\
&\quad \mathsf{isTail}(Tail_2, Y'), \mathsf{union}(Tail_1, Tail_2, Tail), \mathsf{remove}(Y, Y', Y''),\\
&\quad \mathsf{append}(Y'', [o|Tail], Z).\\
\mathsf{binReplyAdd}(X, Y, Y) &\leftarrow \mathsf{isHead}(o, X), \mathtt{objSet}(O, Y), \neg\mathsf{member}(o, O).
\end{aligned}$$

Rules for releaseFW**(flowE,m,flowE')**

$$\begin{aligned}
\mathsf{releaseFW}(X, m, X') &\leftarrow \mathsf{iW}(m, IW), \mathsf{invokeAdd}(RW, X, X').
\end{aligned}$$

$$\begin{aligned}
\mathsf{invokeAdd}([\,], Y, Y) &\leftarrow .\\
\mathsf{invokeAdd}([Head|Tail], Y, Y') &\leftarrow \mathsf{invokeAdd}(Tail, Y, Y''), \mathsf{binInvokeAdd}(Head, Y'', Y').
\end{aligned}$$

$$\begin{aligned}
\mathsf{binInvokeAdd}([\,], Y, Y) &\leftarrow .\\
\mathsf{binInvokeAdd}(X, Y, Z) &\leftarrow \mathsf{isHead}(o, X), \mathsf{member}(Y', Y), \mathsf{isHead}(o, Y'), \mathsf{isTail}(Tail_1, X),\\
&\quad \mathsf{isTail}(Tail_2, Y'), \mathsf{union}(Tail_1, Tail_2, Tail), \mathsf{remove}(Y, Y', Y''),\\
&\quad \mathsf{append}(Y'', [o|Tail], Z).\\
\mathsf{binInvokeAdd}(X, Y, Y) &\leftarrow \mathsf{isHead}(o, X), \mathtt{objSet}(O, Y), \neg\mathsf{member}(o, O).
\end{aligned}$$

Rules for objSet**(X,Y)**

$$\begin{aligned}
\mathtt{objSet}([\,], [\,]) &\leftarrow .\\
\mathtt{objSet}(X, [Head|Tail]) &\leftarrow \mathtt{objSet}(X', [Tail]), \mathsf{isHead}(o, Head), \mathsf{append}(X', o, X).
\end{aligned}$$

Rules for eRdSet**(X,Y)**

$$\begin{aligned}
\mathsf{eRdSet}([\,],[\,]) &\leftarrow .\\
\mathsf{eRdSet}(X,[Head|Tail]) &\leftarrow \mathsf{eRdSet}(X',[Tail]),\mathsf{isTail}(X'',Head),\mathsf{intersection}(X',X'',X).
\end{aligned}$$

Rules for listOfSafeFlow**((x,e),X,a)**

$$\begin{aligned}
\mathsf{listOfSafeFlow}((x,e),[(y,e)],a) &\leftarrow \mathsf{safeFlow}((x,e),[(y,e)],+a)\\
\mathsf{listOfSafeFlow}((x,e),X,a) &\leftarrow \mathsf{isHead}((y,e),X),\mathsf{isTail}(T_X,X),\mathsf{safeFlow}((x,e),[(y,e)],+a),\\
&\quad\ \mathsf{listOfSafeFlow}((x,e),T_X,a)
\end{aligned}$$

Rules for listOfSafeFlow***(X,Y,E,a)**

$$\begin{aligned}
\mathsf{listOfSafeFlow}*([\,],[\,],[\,],+a) &\leftarrow\\
\mathsf{listOfSafeFlow}*(X,E,+a) &\leftarrow \mathsf{isHead}(H_X,X),\mathsf{isHead}(H_Y,Y),\mathsf{isHead}(H_E,E),\\
&\quad\ \mathsf{isTail}(T_X,X),\mathsf{isTail}(T_Y,Y)\mathsf{isTail}(T_E,E),\\
&\quad\ \mathsf{safeFlow}*(H_Y,H_E,+a),\mathsf{isHead}(H_X,H_Y),\mathsf{listOfSafeFlow}*(T_X,T_Y,T_E,a)
\end{aligned}$$

Rules for listOfMtd**(X,M)**

$$\begin{aligned}
\mathsf{listOfMtd}([\,],[\,]) &\leftarrow\\
\mathsf{listOfMtd}(X,M) &\leftarrow \mathsf{isHead}((x,e),X),\mathsf{isHead}(m,M),\mathsf{mtd}(x,m),\\
&\quad\ \mathsf{isTail}(T_X,X),\mathsf{isTail}(T_M,M),\mathsf{listOfMtd}(T_X,T_M)
\end{aligned}$$

Rules for listOfReleaseBK**(X,M,Y)**

$$\begin{aligned}
\mathsf{listOfReleaseBK}([\,],[\,],[\,]) &\leftarrow\\
\mathsf{listOfReleaseBK}(X,M,Y) &\leftarrow \mathsf{isHead}(H_X,X),\mathsf{isHead}(m,M),\mathsf{isHead}(H_Y,Y),\\
&\quad\ \mathsf{isTail}(T_X,X),\mathsf{isTail}(T_M,M),\mathsf{isTail}(T_Y,Y),\\
&\quad\ \mathsf{releaseBK}(H_X,m,H_Y),\mathsf{listOfReleaseBK}(T_X,T_M,T_Y)
\end{aligned}$$