Incorporating Dynamic Constraints in the Flexible Authorization Framework

Shiping Chen, Duminda Wijesekera, and Sushil Jajodia

Center for Secure Information Systems, George Mason University,
Fairfax, VA 22030-4444, USA
{schen3,dwijesek,jajodia}@gmu.edu

Abstract. Constraints are an integral part of access control policies. Depending upon their time of enforcement, they are categorized as static or dynamic; static constraints are enforced during the policy compilation time, and the dynamic constraints are enforced during run time. While there are several logic-based access control policy frameworks, they have a limited power in expressing and enforcing constraints (especially the dynamic constraints). We propose dynFAF, a constraint logic programming based approach for expressing and enforcing constraints. To make it more concrete, we present our approach as an extension to the *flexible authorization framework (FAF)* of Jajodia et al. [17]. We show that dynFAF satisfies standard safety and liveliness properties of a safety conscious software system.

1 Introduction

Constraints are a powerful mechanism for specifying high-level organizational policies [21]. Accordingly, most access control policies contain constraints, usually categorized as static or dynamic, referring to their time of enforcement by the access controller. As examples, consider the following two constraints: *an undergraduate student should not be permitted to grade qualifying examinations at the PhD level*, and *an author should not be allowed to review his/her own manuscript*. The first constraint can be enforced by prohibiting *grading* permissions on PhD examinations for every undergraduate student, thereby making it statically enforceable. The second constraint requires an access controller to evaluate if the requesting subject is also an author of the document to be reviewed when the request is made. This constraint cannot be evaluated prior to the request, making the constraint dynamically, but not statically, enforceable. Enforcing the latter kind of constraints over access permissions expressed as Horn clauses is the subject matter of this paper.

The past decade has seen several logic based *flexible* access control policy specification frameworks. Woo and Lam [25] propose the use of *default logic* for representing access control policies. To overcome the problems of undecidability and non-implementability that arise in Woo and Lam's approach, Jajodia et al. [17] propose an access control policy specification framework (FAF) based on a restricted class of logic programs, viz., those that are *locally stratifiable*.

P. Samarati et al. (Eds.): ESORICS 2004, LNCS 3193, pp. 1–16, 2004.

Bertino et al.'s framework [6] uses *C-Datalog* to express various access control policies [6]. Barker and Stuckey use *constraint logic programming* for multi-policy specification and implementation [4].

Although they are powerful in expressing access control policies, these frameworks have a limited power in specifying and enforcing constraints. For instance, Jajodia et al. [17] use an integrity rule (a logic rule with an `error`() head) to specify constraints. Barker and Stuckey [4] define some special consistency checking rules (with head of predicates *inconsistent_ssd, inconsistent_dsd*) to encode the separation of duty constraints. However, the enforcement of the constraints is left outside the framework; as a result, dynamic constraints cannot be enforced in the access control engine properly.

To overcome these drawbacks, we propose a constraint logic programming based approach to express and enforce dynamic constraints. To make it more concrete, we present our approach as an extension to *Flexible Authorization Framework (FAF)* proposed by Jajodia et al. [17]. Our approach is applicable to other logic based access control frameworks because our constraint specification and enforcement modules are built on top of the existing framework modules. The proposed extension, called *dynFAF*, has two extra modules. First module, the *integrity constraint specification and derivation module (ISM)*, is responsible for specifying the atomic conflicts and deriving all possible complex conflicts in the system that represent the constraints. The second module, the *dynamic access grant module (DAM)*, is responsible for enforcing the constraints specified by ISM dynamically. In addition, DAM allows subjects to *relinquish* permissions that were granted to them. In our design, FAF composes the *static* component, and ISM and DAM compose the *dynamic* component of dynFAF.

We show that dynFAF satisfies safety and liveliness properties granting any access that does not violate derivable constraint, and denying those that do. Because FAF policies are stratified logic programs, they have a stable model semantics [14]. Our constraint specification policies taken together with FAF policies also have a local stratification, thereby admitting a stable model that extends the former. In addition, proposed dynamic access grant module enriches the syntax of the former by having yet another layer of constrained logic programs, that taken as a whole extends the former local stratification. Therefore, a dynFAF specification admits a well-founded model in which some predicate may result in an *undefined* truth in addition to the usual *true* or *false* values; however, our design ensures that any access requested of dynFAF returns only *true* or *false*.

The remainder of the paper is structured as follows. Section 2 contains a brief overview of FAF, followed by a description of its limitations. Section 3 presents the architecture of dynFAF, including the descriptions of ISM and DAM modules. Section 4 presents the semantics of dynFAF syntax. Section 5 shows that dynFAF satisfies the traditional safety and liveliness properties, and that the semantics of the granting and relinquishing access rights are enforced properly. Section 6 compares our work to those of others. Section 7 concludes the paper.

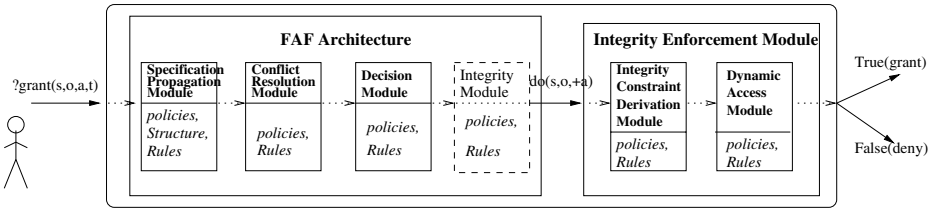

Fig. 1. dynFAF Architecture

2 Overview of FAF

FAF [17] is a logic-based framework to specify authorizations in the form of rules, based on four stages (each stage corresponds to a module) that are applied in a sequence, as shown in FAF Architecture part of Figure 1. In the first stage of the sequence, some basic facts such as authorization subject and object hierarchies (for example directory structures) and a set of authorizations along with rules to derive additional authorizations are given. The intent of this stage is to specify basic authorizations and use structural properties to derive new authorizations. Hence, they are called *specification and propagation policies*. Although propagation policies are flexible and expressive, they may result in *over-specification* resulting in conflicting authorizations. FAF uses *conflict resolution policies* to weed out these in the second stage. At the third stage, *decision policies* are applied in order to ensure the completeness of authorizations. The last stage consists of checking for integrity constraints, where all authorizations that violate integrity constraints will be denied. In addition, FAF ensures that every access request is either granted or rejected, thereby providing a built-in completeness property.

FAF syntax consists of terms that are built from constants and variables (no function symbols) and they belong to four sorts, viz., subjects, objects, actions, and roles. We use the capital letters with subscripts such as X_s, Y_o, X_a, and X_r to denote the respective variables belonging to them, and lower case letters such as `s, a, o`, and `r` for constants. FAF has the following predicates:

1. A ternary predicate `cando`(s, o, a), representing grantable or deniable requests (depending on the sign associated with the action) where s, o, and a are subject, object, and signed action terms, respectively.
2. A ternary predicate `dercando`(s, o, a), with the same arguments as `cando`. The predicate `dercando` represents authorizations derived by the system using inference rules modus ponens plus rule of stratified negation [2].
3. A ternary predicate `do`, with the same arguments as `cando`, representing the access control decisions made by FAF.
4. A 4-ary predicate `done`(s, o, a, t), meaning subject s has executed action a on object o at time t, t is a natural number.
5. Two binary predicate symbols $\texttt{over}_{\texttt{AS}}$ and $\texttt{over}_{\texttt{AO}}$, each taking two subject and object terms as arguments two object terms respectively.

6. A predicate symbol without argument, `error`, symbolizing violation of an integrity constraint, where a rule with an `error` head must not have a satisfiable body.
7. Other terms and predicates necessary to model specific applications. For example, constants AOH, ASH denote object and subject hierarchies with `in`, where `in(x,y,H)` denotes that `x` $\leq$ `y` in hierarchy `H`. For example, we denote the fact that `usr\local` is below `usr` in the object hierarchy AOH by `in(usr\local, usr,` AOH).

Because any FAF specification is a locally stratified logic program, it has a unique stable model [14], and a well-founded model (as in Gelfond and Lifshitz). In addition, the well-founded model coincides with the unique stable model [3, 17]. Furthermore, the unique stable model can be computed in quadratic time data complexity [24]. See [17] for details.

2.1 Limitations of FAF

In its current design, FAF has these limitations. First, FAF expresses constraints using integrity rules of the kind `error`() $\leftarrow L_i, \ldots, L_n$ where `error` is an argument-less predicate that should not be valid in any model and $L_i, \ldots, L_n$ are other literals. Ensuring that granted permissions do not imply `error` is enforced outside of the logical machinery of FAF. Accordingly, it is not within the logical inference engine of FAF to avoid constraint violations. To elaborate further, FAF evaluates an access request as a query ?`do`(s, o, a), and ensures the completeness and consistency of the specification, consisting of the rules from the first three modules, by ensuring that one and only one of `do`$(s, o, +a)$ or `do`$(s, o, -a)$ evaluates to *true*. However, it is possible that both $(s, o, +a)$ and $(s, o, -a)$ could be rejected by the integrity enforcement module making the eventual outcomes incomplete, as the inference rules are unaware of rejections by the integrity enforcement module. Thus, the integrity enforcement needs to be brought inside the reasoning engine, as done in dynFAF.

Second, FAF does not allow constraint derivation, although this occurs in practice. For example, role based access control (RBAC) models have conflicting roles (say, r_1 and r_2) where a single subject assuming them simultaneously violate the policy. In addition, an application may want to declare junior roles of conflicting roles to be conflicting. That is, if roles r_3, r_4 are junior to roles r_1 and r_2, respectively, by satisfying the constrains $r_3 \leq r_1$ and $r_4 \leq r_2$, then no subject should be allowed to assume r_3 and r_4 simultaneously. Our extension facilitates constraint derivation.

Third, in FAF each access request is either granted or denied on its own merit. But some applications may want *controlled (don't care) nondeterminism*. For example, a subject is allowed to assume role r_1 or role r_2, but not both, with no preference for either. If we are to accept a unique stable model, then either r_1 or r_2, but not both can be assumed by the subject. dynFAF facilitates controlled nondeterminism in granting permissions.

Finally, and importantly, FAF does not consider an evolving access control system. That is, if $\texttt{do}(o, s, +a)$ is in the stable model of an authorization specification, the access request (o, s, a) is always permitted. In practice, an authorization may be relinquished by the user or the administrator some time after it is authorized (e.g., in workflow management systems). Consequently, some authorization that is not allowed at a point of time because of constraints restriction may be allowed later if the conflicting authorizations are relinquished. Notice that inserting a negative authorization $\texttt{cando}(o, s, -a)$ does not model this situation. The soon to be described *dynamic access grant module* of dynFAF provides this functionality.

3 dynFAF: A Constraint Logic Programming Based Extension to FAF

To address the problems described in the previous section, FAF is enhanced by adding an *integrity constraint specification and derivation module* (ISM) and a *dynamic access grant module* (DAM), grants accesses that avoid those conflicts. FAF enhanced with these two modules are referred to as *dynFAF*, shown in Figure 1. An access request for (s, o, a) is modeled in dynFAF as a predicate instance $\texttt{request}(s, o, \pm a, t)$, where $(s, o, +a, t)$ is a request to obtain permission for (s, o, a) at time t (equals to a query ?$\texttt{grant}$(s,o,a,t)) and $(s, o, -a, t)$ is a request to relinquish already existing permission for (s, o, a) at time t (equals to a query ?$\texttt{relinquish}$(s,o,a,t)). dynFAF ensures that any request for permission is statically granted by FAF, and granting it does not violate any constraints specified by ISM.

3.1 Integrity Constraint Specification and Derivation Module (ISM)

ISM uses two predicates:

- A binary predicate symbol, `conflict`, where $\texttt{conflict}(\texttt{x},\texttt{y})$ is an atomic conflict where `x` and `y` can be either (subject,object,action) triples or object, action, or subject terms.
- A binary predicate symbol `derConflict`. `derConflict` has the same arguments as `conflict`. $\texttt{derConflict}(\texttt{x},\texttt{y})$ is true iff `x,y` constitute a derivable conflict.

We use Horn clause rules to specify atomic conflicts and derivable conflicts based on already defined atomic conflicts. The corresponding rules are called *conflict rules* and *conflict derivation rules*, respectively. Each conflict rule has a `conflict` predicate as its head and some `cando`, `dercando`, `done` or `rel`-literals as its body.

Example 1 (Conflict Rules)

$$\texttt{conflict}(r_1, r_2) \leftarrow \tag{1}$$

$$\texttt{conflict}(org_A, org_B) \leftarrow \tag{2}$$

$$\texttt{conflict}((o, a), (o', a')) \leftarrow \texttt{isPerm}(o, a), \texttt{isPerm}(o', a'). \tag{3}$$

$$\texttt{conflict}((X_s, X_o, X_a), (X'_s, X_o, X_a)) \leftarrow (X_s \neq X'_s), \texttt{in}(X_s, G, \mathsf{ASH}), \texttt{in}(X'_s, G, \mathsf{ASH}). \tag{4}$$

Rule 1 says that r_1 and r_2 are conflicting roles. Rule 2 says that org_A and org_B are conflicting organizations. Rule 3 says that (o, a) and (o', a') are conflicting permissions. Here the predicate `isPerm(x,y)` *is true if* `x` *is an object and* `y` *is an action. Rule 4 says that any two users in group* `G` *trying to execute the same operation on the same object is a conflict (i.e., only one subject can have the permission at any one time such as a semaphore).*

Each conflict derivation rule has a `derConflict` predicate as its head and some `conflict`, `derConflict`, `cando`, `dercando`, `done`, or `rel`-literals as its body. As it is used recursively, all appearances of `derConflict` literals in the body of conflict derivation rules must be positive. Example 2 shows some conflict derivation rules.

Example 2 (Conflict Derivation Rules)

$$\texttt{derConflict}(X, Y) \leftarrow \texttt{conflict}(X, Y). \tag{5}$$

$$\texttt{derConflict}(X_r, Y_r) \leftarrow \texttt{derConflict}(X_{r'}, Y_{r'}), \texttt{in}(Y_r, Y_{r'}, ASH), \texttt{in}(X_r, X_{r'}, ASH). \tag{6}$$

$$\texttt{derConflict}((X_s, X_o, X_a), (X_s, X'_o, X'_a)) \leftarrow \texttt{conflict}((X_o, X_a), (X'_o, X'_a)). \tag{7}$$

$$\texttt{derConflict}(X_o, Y_o) \leftarrow \texttt{derConflict}(org_A, org_B), \texttt{in}(Y_o, org_B), \texttt{in}(X_o, org_A). \tag{8}$$

$$\texttt{derConflict}((X_s, X_o, read), (X_s, X'_o, read)) \leftarrow \texttt{derConflict}(X_o, X'_o). \tag{9}$$

$$\texttt{derConflict}((X_s, X_r, \texttt{activate}), (X_s, Y_r, \texttt{activate})) \leftarrow \texttt{conflict}(X_r, Y_r). \tag{10}$$

Rule 5 says that every conflict is a derivable conflict. Rule 6 says roles junior to conflicting roles are also conflicting. Rule 7 says that obtaining conflicting permissions leads to a conflict. Rule 8 and rule 9 say that any pair of objects that belong to two conflicting organizations are conflicting objects, and each subject can read only one of them. Rule 10 says activating conflicting roles leads to conflicts.

Note that the conflicts specified in FAF and ISM are different. The former, refers to static compile time conflict, arises out of over specification of permissions, and the latter arises due to application specified conflicts that have to be resolved dynamically by the run time constraint enforcement module DAM. We have chosen to represent only binary conflicts as integrity constraints, inspired by the work such as [16, 5, 20, 15, 19, 20], where most constraints are binary conflicts.

3.2 The Dynamic Access Grant Module (DAM)

As stated earlier, the dynamic access grant module first checks if a requested permission $(s, o, +a, t)$ is permissible under static FAF policies (by checking if $\texttt{do}(s, o, +a)$ is true). If so, then it will further check if granting (s, o, a) at time t would violate any integrity constraints specified in ISM. If the answer is negative, the requested granted and denied otherwise. DAM is based on some assumptions and design choices outlined below.

First, DAM allows requesters to relinquish already granted access rights. Suppose dynFAF granted access request (s, o, a), and after some time the requester s is willing to relinquish the access right (s, o, a). This kind of actions is widely seen in practical systems. (e.g., in workflow management systems) dynFAF allows this kind of actions by modeling it as a query of $?\texttt{relinquish}(s, o, a, t)$ with an insertion of $\texttt{request}(s, o, -a, t)$ into the authorization specification program. Similarly, $\texttt{relinquish}(s, o, a, t)$ has no effect on $\texttt{do}(s, o, \pm a)$ and, therefore, does not alter the unique stable model of the static FAF policy (sans the `error` rules).

Second, we assume that access requests are considered in sequence, *one at a time*, at *discrete time intervals*, referred to as the *synchrony* hypothesis, under which dynFAF accepts only two kinds of requests: to obtain a new permission or to relinquish a permission obtained earlier. They are modelled as inserting instances of $\texttt{request}(s, o, +a, t)$ and $\texttt{request}(s, o, -a, t)$ into the logic program, respectively. dynFAF's answer to requests are computed as an evaluation of $\texttt{grant}(s, o, a, t)$ or $\texttt{relinquish}(s, o, a, t)$, depending upon the chosen sign $\pm$ for the action a in $\texttt{request}(s, o, \pm a, t)$.

Third, we assume that neither $\texttt{grant}(s, o, a, t)$ nor $\texttt{relinquish}(s, o, a, t)$ implies or is implied by $\texttt{done}(s, o, a, t)$ for some time term t. We only assume that whenever an (o, a) is executed by s at a time t, $\texttt{done}(s, o, a, t)$ will be inserted into FAF, satisfying the following conditions: 1) there is a time t' exists such that $\texttt{grant}(s, o, a, t')$ and $t' < t$ hold, 2) there is no t'' exists such that $t'' \in (t', t)$ and $\texttt{relinquish}(s, o, a, t'')$ hold. We call this the *system integrity hypothesis*.

3.3 DAM Syntax

The dynamic access grant module uses five 4-ary predicates `grant`, `relinquish`, `validity`, `holds`, and `request` with arguments subject, object, action and time terms, and a 3-ary predicate `conflictingHold`. The time parameter t here has no relation with the clock time, but serves as a counter of the sequence of requests made to the dynFAF since its inception.

1. $\texttt{grant}(s, o, a, t)$ means that access (s, o, a) is granted at time t.
2. $\texttt{relinquish}(s, o, a, t)$ means that access (s, o, a) is relinquished at time t.
3. $\texttt{validity}(s, o, +a, t)$ means that granting permission (s, o, a) at t does not violate any constraints, while $\texttt{validity}(s, o, -a, t)$ means that granting permission (s, o, a) at time t does violate some constraints.
4. $\texttt{holds}(s, o, a, t)$ means that s holds a access to o at time t.

5. $\texttt{request}(s, o, +a, t)$ means that at time t subject s requests permission (o, a), while $\texttt{request}(s, o, -a, t)$ means that at time t subject s requests to relinquish permission (o, a). Whenever a request (grant or relinquish) is made, the corresponding predicate is inserted into dynFAF as fact.
6. $\texttt{conflictingHold}((s, o, a), (s', o', a'), t)$ means the authorization (s', o', a') is conflicting with (s, o, a) and is holding at time t.

We now specify the rules of DAM that recursively define predicates `grant`, `relinquish`, `holds`, `conflictingHold`, and `validity` as follows:

$$\texttt{grant}(x_s, x_o, x_a, 0) \leftarrow \texttt{request}(x_s, x_o, +x_a, 0), \texttt{do}(x_s, x_o, +x_a). \tag{11}$$

$$\texttt{grant}(x_s, x_o, x_a, x_t + 1) \leftarrow \texttt{validity}(x_s, x_o, +x_a, x_t), \neg\texttt{holds}(x_s, x_o, x_a, x_t), \texttt{request}(x_s, x_o, +x_a, x_t + 1). \tag{12}$$

$$\texttt{relinquish}(x_s, x_o, x_a, x_t + 1) \leftarrow \texttt{holds}(x_s, x_o, x_a, x_t), \texttt{request}(x_s, x_o, -x_a, x_t + 1). \tag{13}$$

Rule 11 says that the first permission that is granted by the dynamic access grant module must be statically permissible and requested. Rule 12 says that any permission that is not already being held, requested and valid (i.e. would not violate any constraints) at time x_t can be granted at time $x_t + 1$. The next set of rules recursively update the `holds` predicate that capture permissions already being held by a subject.

$$\texttt{holds}(x_s, x_o, x_a, x_t) \leftarrow \texttt{grant}(x_s, x_o, x_a, x_t). \tag{14}$$

$$\texttt{holds}(x_s, x_o, x_a, x_t + 1) \leftarrow \texttt{holds}(x_s, x_o, x_a, x_t), \neg\texttt{relinquish}(x_s, x_o, x_a, x_t + 1). \tag{15}$$

Rule 14 says that a permission $(x_o, +x_a)$ granted to x_s at time x_t is held by x_s at time x_t. Rule 15 says that any action other than relinquishing the same permission by itself does not change the holding state of a subject. The following rule defines the predicate `conflictingHold`.

$$\texttt{conflictingHold}((x_s, x_o, x_a), (x'_s, x'_o, x'_a), x_t) \leftarrow \texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)), \texttt{holds}(x'_s, x'_o, x'_a, x_t). \tag{16}$$

The rest of the rules recursively define `validity` that computes if granting a permission to a subject will conflict with the outstanding ones at time x_t.

$$\texttt{validity}(x_s, x_o, +x_a, 0) \leftarrow \neg\texttt{derConflict}((x'_s, x'_o, x'_a), (x_s, x_o, x_a)), \texttt{do}(x_s, x_o, +x_a), . \tag{17}$$

$$\texttt{validity}(x_s, x_o, +x_a, x_t) \leftarrow \texttt{grant}(x_s, x_o, x_a, x_t). \tag{18}$$

$$\texttt{validity}(x_s, x_o, +x_a, x_t) \leftarrow \texttt{relinquish}(x_s, x_o, x_a, x_t). \tag{19}$$

$$\texttt{validity}(x_s, x_o, \pm x_a, x_t + 1) \leftarrow \texttt{grant}(x'_s, x'_o, x'_a, x_t + 1), \texttt{validity}(x_s, x_o, \pm x_a, x_t), \neg\texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)), (x_s, x_o, x_a) \neq (x'_s, x'_o, x'_a). \tag{20}$$

$$\texttt{validity}(x_s, x_o, -x_a, x_t + 1) \leftarrow \texttt{grant}(x'_s, x'_o, x'_a, x_t + 1), (x_s, x_o, x_a) \neq (x'_s, x'_o, x'_a), \texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)). \tag{21}$$

$$\texttt{validity}(x_s, x_o, \pm x_a, x_t + 1) \leftarrow \texttt{relinquish}(x'_s, x'_o, x'_a, x_t + 1), \texttt{validity}(x_s, x_o, \pm x_a, x_t), \neg\texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)), (x_s, x_o, x_a) \neq (x'_s, x'_o, x'_a). \quad (22)$$

$$\texttt{validity}(x_s, x_o, -x_a, x_t + 1) \leftarrow \texttt{validity}(x_s, x_o, -x_a, x_t), \texttt{relinquish}(x'_s, x'_o, x'_a, x_t + 1), \texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)), (x_s, x_o, x_a) \neq (x'_s, x'_o, x'_a), \texttt{holds}(x''_s, x''_o, x''_a, x_t), (x_s, x_o, x_a) \neq (x''_s, x''_o, x''_a), (x'_s, x'_o, x'_a) \neq (x''_s, x''_o, x''_a), \texttt{derConflict}((x_s, x_o, x_a), (x''_s, x''_o, x''_a)). \quad (23)$$

$$\texttt{validity}(x_s, x_o, +x_a, x_t + 1) \leftarrow \texttt{relinquish}(x'_s, x'_o, x'_a, x_t + 1), \texttt{validity}(x_s, x_o, -x_a, x_t), (x_s, x_o, x_a) \neq (x''_s, x''_o, x''_a), \texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)), (x_s, x_o, x_a) \neq (x'_s, x'_o, x'_a), (x'_s, x'_o, x'_a) \neq (x''_s, x''_o, x''_a), \neg\texttt{derConflict}((x_s, x_o, x_a), (x''_s, x''_o, x''_a)). \quad (24)$$

$$\texttt{validity}(x_s, x_o, +x_a, x_t + 1) \leftarrow \texttt{relinquish}(x'_s, x'_o, x'_a, x_t + 1), \texttt{validity}(x_s, x_o, -x_a, x_t), (x_s, x_o, x_a) \neq (x'_s, x'_o, x'_a), \texttt{derConflict}((x_s, x_o, x_a), (x'_s, x'_o, x'_a)), (x_s, x_o, x_a) \neq (x''_s, x''_o, x''_a), (x'_s, x'_o, x'_a) \neq (x''_s, x''_o, x''_a), \neg\texttt{conflictingHold}((x_s, x_o, x_a), (x''_s, x''_o, x''_a), x_t). \quad (25)$$

$$\texttt{validity}(x_s, x_o, -x_a, x_t) \leftarrow \neg\texttt{validity}(x_s, x_o, +x_a, x_t). \quad (26)$$

Rule 17 is the base step of the recursion saying that every permission that is allowed by the static component FAF and there is no conflicting permissions existing is valid when the dynamic grant module begins. The next two rules 18 and 19 state that an authorization (s, o, a) is valid at time t if it is granted or relinquished at time t. Rules 20 and 21 consider how some other (subject,object,action) pair being granted affects the validity of a permission. Rule 20 leaves the validity if a non-conflicting permission was granted at time x_t. Rule 21 invalidates it if another conflicting permission was granted at x_t.

The next four rules address what happens to the state of validity of a permission if some other permissions were relinquished. Rule 22 says that if the relinquished permission is not conflicting with the considered one, then the validity remains the same. Rule 23 says that if there are other conflicting authorizations held in the system, then the validity states of the considered one remains invalid. Rule 24 says that if the relinquished permission is the only conflicting permission with the considered one in the system, the validity state of the considered one will be changed to valid. Rule 25 says that although there are still other conflicting permissions with the considered one, if they are all not holding, then the validity state of the considered one will be valid. The last rule, rule 26 says that $\texttt{validity}(x_s, x_o, -x_a, x_t)$ succeeds, when $\texttt{validity}(x_s, x_o, +x_a, x_t)$ fails. This rule is necessary because of the following reason. In rules 20, 22, 23 and 24, $\texttt{validity}(x_s, x_o, \pm x_a, x_t + 1)$ depends upon $\texttt{validity}(x_s, x_o, \pm x_a, x_t)$.

Thus, in order for the inductive computation of other steps to proceed from x_t to $x_t + 1$ it is necessary to have $\texttt{validity}(x_s, x_o, -x_a, x_t)$ which is not given by rule 21. Therefore, the rule 26 allows us to infer $\texttt{validity}(x_s, x_o, -x_a, x_t)$ from the failure of $\texttt{validity}(x_s, x_o, +x_a, x_t)$.

In section 5 we show that in view of the synchrony hypothesis, `grant`, `relinquish`, `validity` and `holds` are correctly specified. That is `grant` and `relinquish` satisfy both safety and liveliness properties and have the required anti-idempotency properties. In other words, no subject can relinquish permissions that it does not hold and a subject cannot obtain permissions that it already holds.

Example 3 (Interacting with DAM) *Consider the separation of duty example which states that two processes p_1 and p_2 may not get write locks on a file "foo" at the same time, but otherwise both processes have static permissions to write to "foo". These requirements are captured by dynFAF as follows.*

$$\texttt{do}(p_1, \text{“}foo\text{”}, +write) \leftarrow . \tag{27}$$

$$\texttt{do}(p_2, \text{“}foo\text{”}, +write) \leftarrow . \tag{28}$$

$$\texttt{derConflict}((p_1, \text{“}foo\text{”}, write), (p_2, \text{“}foo\text{”}, write)) \leftarrow . \tag{29}$$

Rules 27 and 28 state static write permissions to "foo" is given to p_1 and p_2 respectively. Rule 29 says that, although p_1 and p_2 have static write permissions, they may not use them simultaneously.

Now consider a request to obtain write permission to "foo" by p_1 at time 0. That means, `request`*(p_1,"foo",+write,0) now becomes a part of the DAM rules. Therefore,* `grant`*(p_1,"foo",write,0) now is evaluated to be true as, by rule 11,* `grant`*(p_1,"foo",write,0) holds if* `request`*(p_1,"foo",+write,0) and* `do`*(p_1,"foo",+write) are valid. As a result, p_1 is granted write permission on "foo".*

Now consider a request to write "foo" issued by p_2 at time 1. This request is modeled by entering `request`*(p_2,"foo",+write,1) into dynFAF's rule set. Because* `derConflict`*((p_1,"foo",write),(p_2,"foo",write)) holds, by rule 17 we can not get* `validity`*(p_2,"foo",+write,0). Therefore* `grant`*(p_2,"foo",write,1) cannot become valid as the only applicable rule 12 results in finite failure. Similarly,* `grant`*(p_1,"foo",write,1) also fails in the alternate addition of* `request`*(p_1,"foo",+write,1) instead of* `request`*(p_2,"foo",+write,1) because* `holds`*(p_1,"foo",write,1) is valid by rule 14 and 15.*

Now suppose that p_1 wishes to relinquish its "write" permission to "foo" at time 2. This is done by entering `request`*(p_1,"foo",–write,2) into dynFAF's rule set. Then by rule 13,* `relinquish`*(p_1,"foo",write,2) evaluates to true because of* `holds`*(p_1,"foo",write,1) and* `request`*(p_1,"foo",–write,2). Now suppose p_2 requests write permission to "foo" again by inserting* `request`*(p_1,"foo",write,3). At this time* `grant`*(p_2,"foo",write,3) succeeds by rule 12 as* `validity`*(p_2,"foo",+write,2) holds by rule 24. As a result, request (p_2,"foo",write) is granted at time 3.*

4 Semantics of dynFAF

This section describes models of dynFAF syntax. All lemmas and theorems in this section are given without proof. We refer the reader to [9] for the formal proofs. We consider a dynFAF specification to consist of FAF rules (without the `error` predicates), conflict rules, conflict derivation rules, a collection of appropriately constructed (soon to be described) `request` predicates, and rules from 11 to 26. As argued in [17], FAF policies are locally stratified logic programs. We now show that dynFAF specifications form locally stratified logic programs.

Lemma 1 (Local stratification and stable models of dynFAF). *Any dynFAF specification constitutes a local stratification on its Herbrand base. Consequently, dynFAF rules restricted to FAF and ISM have a unique stable model that coincides with its well-founded model.*

At any time n, a dynFAF specification consists of the following four kinds of rules: FAF rules, ISM rules, a set of n instances of `request` predicates $\{$`request`$(\_,\_,\_,i) : i \leq n\}$, and DAM rules consisting of rules 11 through 26, that we refer to as F, C, Tr_n, D, respectively. Lemma 1 says that $F \cup C$ has a unique stable model, denoted by $\mathcal{M}(F \cup C)$. Now we build a model, notated as $\mathcal{M}(F \cup C \cup Tr_n \cup D)$, for $F \cup C \cup Tr_n \cup D$ for each n and another model $\mathcal{M}(F \cup C \cup (\cup_{i\in\omega} Tr_i) \cup D)$ for $F \cup C \cup (\cup_{i\in\omega} Tr_i) \cup D$ as three-valued Kripke-Kleene (also known as Fitting-Kunen)models [18, 11] over $\mathcal{M}(F \cup C)$. We show that every `grant` or `relinquish` query instance over either of these models evaluate to either true or false, and therefore the three-valued model takes only two truth values. In doing so, we have to choose a constraint domain and interpretation for negation. We choose the finite integer domain, CLP($\mathcal{R}$), as our constraint domain and interpret negation as *constructive* negation [7, 8] as have been used for negation by Fages [13, 10]. The latter choice is due to the almost universally accepted stance on using constructive negation instead of its classical counterpart or negation as failure [22, 23, 13, 10]. The former choice is due to the fact that constructive negation proposed by Stuckey [22, 23] requires that the constraint domain be *admissible closed*, whereas that proposed by Fages [13, 10] does not (at the cost of requiring some uniformity in computing negated subgoals of a goal). We now give formal details. As a notational convention hereafter we denote $\mathcal{M}(F \cup C \cup Tr_n \cup D)$ by $\mathcal{M}(F, C, Tr_n, D)$ and $\mathcal{M}(F \cup C \cup (\cup_{i\in\omega} Tr_i) \cup D)$ by $\mathcal{M}(F, C, Tr_*, D)$.

Definition 1 (n-traces, *-traces and models) *For every numeral n, any set of predicate instances of the form $\{$*`request`$(\_,\_,\_,i) : i \leq n\}$*, $\{$*`request`$(\_,\_,\_,i) : i \in \omega\}$ *are said to be an n-trace and a *-trace respectively, where every* `request`$(\_,\_,\_,i)$ *term is variable free.*

*Let F, C, Tr_n, Tr_*, and D be respectively FAF, ISM, n-trace, *-trace, and DAM rules. Let Φ be the three-valued Kripke-Kleene immediate consequence operator. Then we say that $\bigcup_{i\in\omega} \Phi^i_{R\cup D}(F \cup C)$ are the models $\mathcal{M}(F, C, R, D)$ where R is either an n-trace Tr_n or an *-trace Tr_*.*

As stated in definition 1, a model of $\mathcal{M}(F, C, R, D)$ is obtained by evaluating the Φ operator over a FAF+ISM model $\mathcal{M}(F, C)$ ω many times. The reasoning behind our choice for the semantics is that FAF already has a (two-valued) stable model. In [12], Fitting shows that $\bigcup_{i\in\omega} \Phi^i_{Tr_n \cup D}(F \cup C) = \bigcup_{i\in\omega} \Phi^i_{Tr_n \cup D, F\cup C}(\emptyset)$. The dynamic grant policy then extends this stable model. Following two claims clarify this statement.

Theorem 1 (Finite termination of dynFAF queries).
For every numeral n, every `grant`$(\_, \_, \_, n)$, `relinquish`$(\_, \_, \_, n)$, `holds`$(\_, \_, \_, n)$, *and* `validity`$(\_, \_, \_, n)$ *query either succeeds or fails finitely, given that query over $\mathcal{M}(F, C)$ has the same property.*

Consequently, for every numeral n, the three valued model $\mathcal{M}(F, C, R, D)$ evaluates every instance of `grant`$(\_, \_, \_, n)$ *or* `relinquish`$(\_, \_, \_, n)$ *query to be either true or false.*

Theorem 1 shows that dynFAF acts like FAF in the sense that every request is either honored or rejected. But theoretically there is a remarkable difference between a FAF model and a dynFAF model. While every FAF model is a (least) fixed point of a monotonic operator (conventionally referred to as the classical immediate consequence operator **T**), a dynFAF model is not a fixed point of the so called *Fitting-Kunen* Φ operator [12, 10], as it is well known that the closure ordinal of the Fitting-Kunen Φ operator is not ω. ([12] gives a simple counterexample) In contrast, a dynFAF model $\mathcal{M}(F, C, R, D)$ is an ω-closure of the Φ operator of $\mathcal{M}(F, C)$ under rules 11 through 26.

The other pertinent point is that nothing in the logical machinery guarantees that the synchrony hypothesis is valid, but conversely the correctness of the dynamic access grant module depends upon this externally enforced assumption. The next set of results show the connections between different traces.

Definition 2 (Trace chains) *We say that a set of n-traces $\{Tr_n : n \geq 0\}$ is a trace chain iff each Tr_n is an n-trace and $Tr_n \subset Tr_{n+1}$. Then we say that $Tr_* = \bigcup_{i\in\omega} Tr_i$ is the limit of the trace set $\{Tr_n : n \geq 0\}$.*

Following results are valid for any trace chain.

Lemma 2 (Compactness of Traces).
Suppose $\{Tr_n : n \geq 0\}$ is a trace chain. Then the following holds:

1. *$\mathcal{M}(F, C, Tr_n, D) \models \phi$ iff $\mathcal{M}(F, C) \models \phi$ for every FAF or ISM predicate instance ϕ.*
2. *$\mathcal{M}(F, C, Tr_n, D) \models \phi$ iff $\mathcal{M}(F, C, Tr_{n+1}, D) \models \phi$ for every DAM predicate instance where the last variable of ϕ is instantiated to m for any $m \leq n$.*
3. *$\mathcal{M}(F, C, Tr_*, D) \models \phi$ iff $\mathcal{M}(F, C, Tr_n, D) \models \phi$ where ϕ is a variable free DAM predicate where the numeral instance is n.*

Lemma 2 says that any model $\mathcal{M}(F, C, Tr_n, D)$ of $F \cup C \cup Tr_n \cup D$ only validates the history of dynamic changes taking place over the static model $\mathcal{M}(F, C)$

up to and including time n. It also says that evaluating the Fitting-Kunen Φ closure operator ω many times does not add any more *truth* to $\mathcal{M}(F, C, Tr_n, D)$ than n many times. In that respect, our semantics is finite over finite lifetimes of dynFAF evolutions.

5 Correctness of DAM

This section shows that dynFAF functions correctly. All lemmas and theorems in this section are given without proof. We refer the reader to [9] for the formal proofs. Our notion of correctness consists of two parts: (1) dynFAF satisfies traditional safety and liveliness properties. (2) `grant` and `relinquish` function as expected. By *safety* we mean that any granted permission does not violate any constraint. By *liveliness*, we mean that any requested permission that does not conflict with any other outstanding permissions is granted. By the expected functionality of `grant` we mean that any request for already granted permission fails. Similarly, the expected functionality of `relinquish` is that only granted permissions are relinquishable. In order to prove these results, we prove Lemma 3, that guarantees the correctness of `holds` and `validity`, the two other predicates that are for *internal* use in DAM.

Lemma 3 (Correctness of `holds` and `validity`).
The following statements hold for every numeral n and every permission triple (s, o, a):

1. $\mathcal{M}(F, C, Tr_n, D) \models \texttt{holds}(s, o, a, n)$ *iff* $\mathcal{M}(F, C, Tr_n, D) \models \texttt{grant}(s, o, a, n')$ *for some* $n' \leq n$ *and* $\mathcal{M}(F, C, Tr_n, D) \not\models \texttt{relinquish}(s, o, a, m)$ *for all* m *satisfying* $n' < m \leq n$.
2. $\mathcal{M}(F, C, Tr_n, D) \models \texttt{validity}(s, o, +a, n)$ *iff there is no permission* (s', o', a') *satisfying* $\mathcal{M}(F, C, Tr_n, D) \models \texttt{holds}(s', o', a', n) \wedge \texttt{derConflict}((s, o, a), (s', o', a')) \wedge \texttt{do}(s', o', +a')$.

Now we use this result to prove the safety and the liveliness as promised.

Theorem 2 (Safety and liveliness of dynFAF).
The following holds for all permissions (s,o,a) and all times n:

Safety: *Suppose* $\mathcal{M}(F, C, Tr_n, D) \models \texttt{holds}(s, o, a, n)$. *Then there is no other permission* (s', o', a') *satisfying* $\mathcal{M}(F, C, Tr_n, D) \models \texttt{holds}(s', o', a', n) \wedge \texttt{derConflict}((s, o, a), (s', o', a'))$.

Liveliness: *Suppose* $\mathcal{M}(F, C, Tr_n, D) \models \texttt{validity}(s, o, a, n) \wedge \neg\texttt{holds}(s, o, a, n)$. *Then there is a trace* Tr_{n+1} *satisfying* $\mathcal{M}(F, C, Tr_{n+1}, D) \models \texttt{grant}(s, o, a, n+1) \wedge \texttt{holds}(s, o, a, n+1)$.

Now we show that `grant` and `relinquish` has the required prerequisites. That is, $\texttt{request}(\_, \_, (+)\_, n)$ succeeds and results in $\texttt{grant}(\_, \_, \_, n)$ evaluating to *true* only if this permission has not already outstanding. Similarly, $\texttt{request}(\_, \_, (-)\_, n)$ succeeds and results in $\texttt{relinquish}(\_, \_, \_, n)$ evaluating to *true* only if this permission is already outstanding.

Theorem 3 (Prerequisites of `grant` and `relinquish`).
The following holds for all (s,o,a) and all n:

`grant`: $\mathcal{M}(F, C, Tr_{n+1}, D) \models \texttt{grant}(s, o, a, n+1)$ *only if* $\mathcal{M}(F, C, Tr_n, D) \not\models \texttt{holds}(s, o, a, n)$.
`relinquish`: $\mathcal{M}(F, C, Tr_{n+1}, D) \models \texttt{relinquish}(s, o, a, n+1)$ *only if* $\mathcal{M}(F, C, Tr_n, D) \models \texttt{holds}(s, o, a, n)$.

6 Related Work

Ahn and Sandhu introduce a logical language for specifying role-based authorization constraints named *RCL2000* [1]. They identify conflicts as originating from conflicting permissions, users and roles, and constraints are stated using cardinalities of sets of access or their intersections where most cardinalities are restricted to one. They specify several kinds of *dynamic separation of duty*, without showing enforcement mechanisms.

Bertino et al. propose a framework for specification and enforcement of authorization constraints in workflow management systems [5]. They present a language to express authorization constraints as clauses in a logic program and propose algorithms to check for the consistency of the constraints and to assign roles and users to the workflow tasks in such a way that no constraints are violated. The consistency checking algorithms is executed by the security officer in role or user planning.

Similar to FAF, Barker and Stuckey [4] define some special consistency checking rules (with head of predicates *inconsistent_ssd, inconsistent_dsd*) to encode the separation of duty constraints. The constraints are checked by the security officer whenever new user-role or new role-permission assignments are inserted.

In comparison, dynFAF has the following advantages in expressing and enforcing constraints. First, we add the ability to derive all binary constraints from atomic constraints specified by the SSO using predefined derivation rules. Second, derived constraints are enforced dynamically in the model itself. That is, all those and only those access requests that do not violate any constraint will be granted – referred to as liveliness and safety, respectively.

7 Conclusions

In this paper, we described a constraint logic programming-based approach, dynFAF, for expressing and enforcing dynamic constraints as an extension to the framework FAF proposed by Jajodia et al. [17]. We limited FAF to rules without `error` predicates; then we enriched FAF with the ability to specify atomic binary conflicts and derive complex conflicts using user specified rules. This extension constitutes the ISM module. We showed how to extend this syntax with an appropriate dynamic access grant module DAM. DAM grants requests that do not violate any conflicts. In return, DAM expects the user of the system to relinquish granted permissions once they are no longer in need. dynFAF works

under the assumptions that access requests are submitted in sequence, one at a time and that the permissions obtained are relinquished by giving them up to the access controller. The current design of dynFAF requires that these are enforced external to the system.

We have shown that dynFAF models have a unique three-valued model used in (constraint) logic programming. We have further shown that any stable model (correspondingly well-founded) FAF model is extendible to a three-valued dynFAF model. In addition, we showed that every instance of a request to grant or relinquish permissions made to dynFAF always terminates without floundering as a constraint logic program.

Acknowledgments

This work was partially supported by the National Science Foundation under grant CCR-0113515 and IIS-0242237. We thank the anonymous reviewers for their valuable comments.

References

1. G. Ahn and R. Sandhu. Role-based authorization constraints specification. *ACM Transactions on Information and Systems Security*, 3(4):207–226, November 2000.
2. K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.
3. C. Baral and V. S. Subrahmanian. Stable and extension class theory for logic programs and default theories. *Journal of Automated Reasoning*, 8(3):345–366, June 1992.
4. S. Barker and P. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transactions on Information and System Security*, 6(4):501–546, 2004.
5. E. Bertino and V. Atluri. The specification and enforcement of authorization constraints in workflow management. *ACM Transactions on Information Systems Security*, 2(1):65–104, February 1999.
6. E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, February 2003.
7. D. Chan. Constructive negation based on the completed databases. In R. A. Kowalski and K. A. Bowen, editors, *Proc. International Conference on Logic Programming (ICLP)*, pages 111–125. The MIT Press, 1988.
8. D. Chan. An extension of constructive negation and its application in coroutining. In E. Lusk and R. Overbeek, editors, *Proc. North-American Conference on Logic Programming*, pages 477–489. The MIT Press, 1989.
9. S. Chen, D. Wijesekera, and S. Jajodia. Incorporating dynamic constraints in the flexible authorization framework. Technical Report CSIS-TR-04-01, Center for Secure Information Systems, George Mason University, June 2004.
10. F. Fages. Constructive negation by pruning. *Journal of Logic Programming*, 32(2):85–118, 1997.

11. M. Fitting. A kripke-kleene semantics for logic programs. *Journal of Logic Programming*, 2(4):295–312, 1985.
12. M. Fitting and M. Ben-Jacob. Stratified, weak stratified, and three-valued semantics. *Fundamenta Informaticae, Special issue on LOGIC PROGRAMMING*, 13(1):19–33, March 1990.
13. F. Francois and G. Roberta. A hierarchy of semantics for normal constraint logic programs. In *Algebraic and Logic Programming*, pages 77–91, 1996.
14. M. Gelfond and L. Lifschitz. The stable model semantics for logic programming. In *Proc. Fifth International Conference and Symposium on Logic Programming*, pages 1070–1080, 1988.
15. T. Jaeger. On the increasing importance of constraints. In *Proc. of the Fourth Role Based Access Control*, pages 33–42, Fairfax, VA, 1999.
16. T. Jaeger, A. Prakash, J. Liedtke, and N. Islam. Flexible control of downloaded executable content. *ACM Transactions on Information Systems Security*, 2(2):177–228, May 1999.
17. S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):214–260, June 2001.
18. K. J. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(4):298–308, December 1987.
19. M. Nayanchama and S. Osborn. The role graph model and conflict of interest. *ACM Transactions on Information and Systems Security*, 2(1):3–33, February 1999.
20. S. Osborn, R. Sandhu, and Q. Munawer. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Transactions on Information and Systems*, 3(2):85–106, May 2000.
21. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Febraury 1996.
22. P. Stuckey. Constructive negation for constraint logic programming. In *Logic in Computer Science*, pages 328–339, 1991.
23. P. Stuckey. Negation and constraint logic programming. *Information and Computation*, 118(1):12–33, 1995.
24. A. van Gelder. The alternating fixpoint of logic programs with negation. In *Proc. 8th ACM Symposium on Principles of Database Systems*, pages 1–10, 1989.
25. T. Y. C. Woo and S. S. Lam. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, 2(2-3):107–136, 1993.