Specification and Analysis of Dynamic Authorisation Policies

Moritz Y. Becker
Microsoft Research, Cambridge, CB3 0FB, United Kingdom
moritzb@microsoft.com

Abstract

This paper presents a language, based on transaction logic, for specifying dynamic authorisation policies, i.e., rules governing actions that may depend on and update the authorisation state. The language is more expressive than previous dynamic authorisation languages, featuring conditional bulk insertions and retractions of authorisation facts, non-monotonic negation, and nested action definitions with transactional execution semantics. Two complementary policy analysis methods are also presented, one based on AI planning for verifying reachability properties in finite domains, and the second based on automated theorem proving, for checking policy invariants that hold for all sequences of actions and in arbitrary, including infinite, domains. The combination of both methods can analyse a wide range of security properties, including safety, availability and containment.

1. Introduction

Security policy should be separated from its actual enforcement. In the context of access control, this allows the authorisation policy to be configured without requiring costly and error-prone modifications of the reference monitor. To facilitate the separation, a multitude of authorisation languages and logics has been developed over the past 15 years (e.g. [1], [10], [19], [31], [28], [32], [9], [7], [24]).

However, despite the increasing expressiveness of such languages, there is an important class of authorisation policies that most languages cannot adequately express: *dynamic* policies, or policies governing actions that both depend on and update the authorisation state. Consider, for instance, the following policy rule for initiating a cheque payment (written in SecPAL [7]):

$$\texttt{canInitPayment}(x,p):$$
$$\quad \texttt{Bank says } x \texttt{ is a manager},$$
$$\quad \neg\exists y(\texttt{Bank says } y \texttt{ has initiated } p)$$

According to this policy fragment, managers can initiate a payment p that has not been already initiated. Therefore, the *precondition* for granting the action checks for the absence of has initiated facts in the authorisation state. But for this scheme to work as intended, a granted payment initiation action must have the side effect of inserting a has initiated fact into the state. As such effects cannot be expressed in the language, it has to be hard-coded into the reference monitor, as illustrated in pseudocode below.

```
handleRequests(Action a, Params ps) {
 if (Policy.permits(a, ps) {
  if (a=='canInitPayment') {
   Policy.insert(ps[0] has initiated ps[1]);
  } [...]
  execute(a, ps);
 } [...]
}
```

As dynamic updates of the authorisation state are arguably part of the policy (and may be subject to frequent policy changes), we see that the separation of policy and enforcement mechanism has not been fully achieved.

This paper proposes a new, *dynamic*, authorisation logic, DYNPAL, to help factoring out both the static and dynamic aspects of a policy from the reference monitor. The logic has a number of unique features:

- Parameterised predicates with variables ranging over a possibly infinite domain. This is crucial in decentralised settings, where the set of principals and objects are generally unbounded.
- Negated predicates with negation as failure semantics (as opposed to standard or linear negation), which is needed for testing the absence of facts in the authorisation state, e.g. in separation of duty constraints.
- Two primitives for updates: conditional bulk insertions and retractions of authorisation facts of the form $\pm\{A_1 : A_2\}$ (insert/retract all facts of the form A_1 such that A_2 holds), e.g. for expressing cascading revocation.
- Actions, updates and purely static conditions are treated uniformly as logical literals and can be interspersed in any order, to express not just preconditions, but also postconditions and intermediate conditions. This is useful e.g. for expressing integrity constraints. Action definitions can be nested, leading to more succinct and modular policies.
- Transactional execution semantics (based on transaction logic [12], [11]) ensuring that actions with

multiple updates are executed atomically (either all or none are committed).

This feature set facilitates a wider range of dynamic policies than previous dynamic languages (such as [9], [14], [8]), as illustrated in Section 4.

The consequences of a dynamic policy are generally much harder to foresee than those of a purely static policy. With dynamic policies, it is not sufficient to consider a single action; rather, we have to consider *sequences* of actions of arbitrary length, performed by a possibly infinite coalition of agents. The increased complexity calls for computer-assisted techniques for policy debugging and verification. There has been some work on analysing dynamic policies (see Section 2), but only on less expressive authorisation languages and only for fixed, finite coalitions of agents. Apart from the design of DYNPAL, the main technical contributions of this paper are novel automated reasoning techniques for solving two complementary problems in analysing DYNPAL policies.

The first problem is *reachability*: given a policy, a domain (of involved principals and other objects) and an initial state, is there a sequence of actions that leads to a state satisfying the goal property? For infinite domains, this problem is shown to be undecidable. For finite domains, an algorithm is presented for translating a reachability query into a PDDL specification [22], a language for describing AI planning [21], [44] problems. The main benefit of this approach is that it avoids the difficulties of devising and implementing a dedicated, efficient search algorithm. As AI planning has matured over decades, highly optimised goal-directed heuristics to avoid constructing the entire model, as implemented in existing AI planners (such as FF [26]), can be effectively leveraged.

The second problem is that of checking *policy invariants*, safety properties that remain unaffected under any sequence of actions of arbitrary length and in any arbitrary domain, including *unbounded* and *infinite* ones (for example, the property that, in a conference system, authors may never review their own papers, even if an unbounded number of principals are collaborating). This paper presents a translation from a given policy and an invariance hypothesis into a validity problem of first order logic (FOL) over finite structures. Again, this solution enables the use of existing tools, in this case FOL theorem provers (such as Prover9-Mace4 [37]).

The two analyses are complementary: if reachability analysis fails to produce a sequence of actions leading to the specified (safe or unsafe) goal state, invariance analysis may be able to prove that no such sequence exists in any domain. Vice versa, if the invariance hypothesis is proven wrong, reachability analysis may provide a specific sequence of actions that violates the property. Both analyses together cover a wide range of security queries (including effective permissions, safety, availability, containment and correspondence queries, in both finite and infinite domains).

The remainder of the paper is organised as follows. Related work on previous dynamic authorisation languages and on dynamic policy analysis is reviewed in Section 2. DYNPAL's syntax is introduced in Section 3. Section 4 provides examples of dynamic policies written in the language and motivates the design choices behind it. A formal semantics and a proof system are given in Section 5. The two analysis methods are described in Sections 6 and 7. The paper concludes with a discussion of experimental results and limitations in Section 8.

2. Related work

Dynamic authorisation languages. SMP [8] is a precursor of DYNPAL, but lacks conditional bulk updates, stratified negation, and nested action definitions. A memoing backward-chaining algorithm is used for analysing reachability in finite domains. Cassandra [9] is an authorisation language that defines the actions of activating a role and deactivating a role, along with a transition system that updates the authorisation state by inserting and retracting corresponding 'hasActivated' facts. Users can thus write state-dependent and implicitly state-updating access control policies, but this rather ad-hoc approach is inflexible and not very user-friendly. In a similar spirit, dynFAF [14] keeps track of the history of user requests by dynamically adding facts (with a time-stamp parameter) to the logic program. In dynFAF, facts are never removed; instead, permissions are signed, and permission revocation is modelled by adding a fact with a negative permission. Jagadeesan et al. [27] use a sub-language of Timed Default Concurrent Constraint Programming for modelling dynamic policies. Their language, being almost a full-fledged procedural programming language, can express state changes triggered by both user requests as well as environmental changes. This high expressiveness comes at a price: policies are generally harder to analyse, and evaluation may not terminate.

Dynamic policy analysis. To the best of my knowledge, this paper is the first to study *invariance* analysis of dynamic policies in the context of unknown, unbounded or infinite domains. The rest of this section focusses on prior work on *reachability* analysis for dynamic policies, which is also studied in this paper.

The earliest dynamic model of access control probably is HRU [25]. The state is represented as a standard access control matrix. Actions are preconditioned on the state and may define a series of elementary updates: modifying cells in the matrix and adding or removing rows or columns (corresponding to creating or destroying principals or re-

sources). The reachability problem in HRU is undecidable. HRU can be modelled in DYNPAL with infinite domain, which would provide an alternative proof of the current paper's undecidability result (Proposition 6.5).

Li et al. [33] study role-based policies (written in RT [32]) where role-membership rules may be added or removed by principals, under specifiable restrictions (e.g. that membership for certain roles may only grow or shrink). This induces a system where role membership can change dynamically. They investigate the complexity of various specific queries in such a system, including role membership reachability ('*can P ever become a member of role R?*') and containment ('*does every principal having role R_1 always also have role R_2?*'). This is one of the very few examples of prior work that does not assume the set of principals to be known and finite. Apart from that, the setting in their work is rather different from the current paper's, in that the RT fragments they consider are of relatively limited expressiveness, and they do not study state-changing actions governed by policy.

Bandara et al. [3] model Ponder [15] policies with obligations (which trigger state changes) in the Event Calculus [29]. They encode potential policy bugs (such as breach of separation of duty) as goals, and use abduction over the Event Calculus policy to find sequences of actions that satisfy the goal. Only finite domains are considered.

Sasturkar et al. [40] analyse reachability and availability properties in Administrative Role-Based Access Control (ARBAC [39]) policies. In terms of expressiveness, ARBAC is roughly equivalent to DYNPAL rules without parameters and variables, where actions may only have *pre*conditions, and each action may only either add a single user-role membership or revoking it. Reachability and availability analysis is proven to be PSPACE-complete; as in the current paper, their result is proven by establishing a connection to AI planning. If negative preconditions are disallowed, the analysis becomes tractable, and similarly when role revocation and disjunctive preconditions are disallowed. Only finite ARBAC models are considered. The computational complexity of ARBAC is further studied by Stoller et al. [42], showing that mixed roles (roles that appear both as positive and negative preconditions) are mainly responsible for intractability. The problem becomes polynomial in the input size when the number of mixed roles is kept constant. They also present a plan searching algorithm for ARBAC. It would be interesting to see how its performance compares with general-purpose planners.

Dougherty et al. [16] study reachability, availability and containment queries for dynamic access control policies where the static part of the policy is specified in Datalog (but without negative preconditions), but state updates are not specified in a language; rather, they are represented as transitions in a finite state machine. This way, environmental changes not effected by a user can be modelled as well. They prove that reachability is in NLOGSPACE via an encoding in linear temporal logic. Subsequent work by the same authors [17] focusses on policies with obligations which they abstractly define as stateful constraints on execution paths. They explore a wide range of analyses using Büchi automata, but again only in finite domains.

Barth and Mitchell [4] focus on stateful policies for digital rights management which, for example, govern how often a song may be played on a device. They investigate a problem that is quite different from our reachability problem, namely deciding if a particular sequence of actions is authorised. This question is non-trivial in their setting due to nondeterminism: exercising a right may consume a number of different licenses to choose from. They show that this problem is NP-complete (for finite domains), and propose an algorithm for evaluating sequences of actions based on propositional linear logic that is monotonic in the sense that more flexible digital licenses also permit more actions.

In the RW framework [45], a state is a set of propositional variables, and actions may only set or unset one specific state variable. Preconditions are expressed in propositional logic (which is sufficient as they only consider finite domains). Postconditions, intermediate conditions, or nested actions are not supported by RW (or any other modelling mechanism cited in this section). RW also includes a complex query language for sequences of actions that allows specification of coalitions and nested goals; for example, it is possible to ask "*can Alice and Bob together perform actions that leads to a state where Charlie can perform some actions such that eventually Doris can delete a file?*". RW can also specify preconditions for reading the value of a property. This plays a role in their distinction between *strategies* and *guessing strategies*. In the former, only those actions are allowed where the acting principal *knows* (by having read the relevant properties) that she is permitted to perform them. Guessing strategies do not have this restriction. The authors have developed their own model checking tool, AcPeg, which, however, only works for finite domains.

3. DYNPAL Syntax

DYNPAL is designed to be minimalistic, and focusses specifically on the dynamic aspects of authorisation policies; as such, it does not include some features that would be useful in a complete authorisation language such as parameter types, delegation primitives, variable constraints and a natural-language-like syntax. It is designed such that its primitives for expressing dynamic state updates can be easily added to other logic-based authorisation languages (such as SecPAL [7], see Section 8).

Static literals	$\langle L^{st} \rangle$	::=	$A^{st} \mid \neg A^{st}$
Dynamic literals	$\langle L^{dy} \rangle$	::=	A^{dy}
		$\mid$	$+\{p^{ex}(\vec{x}) : A^{st}\}$
		$\mid$	$-\{p^{ex}(\vec{x}) : A^{st}\}$
Literals	$\langle L \rangle$	::=	$L^{st} \mid L^{dy}$
Static rules	$\langle R^{st} \rangle$	::=	$A^{in} \leftarrow \vec{L}^{st}$
Dynamic rules	$\langle R^{dy} \rangle$	::=	$A^{dy} \leftarrow \vec{L}$

Figure 1. Above, $\vec{x}$ is a sequence of distinct variables, $p^{ex} \in \mathbf{Pred}^{ex}$, $A^{st} \in \mathbf{Atom}^{st}$, $A^{dy} \in \mathbf{Atom}^{dy}$, $A^{in} \in \mathbf{Atom}^{in}$. A dynamic literal of the form $\pm\{A_1 : A_2\}$ is an *elementary update*, and A_2 is its *guard*. In a rule, the atom to the left of the arrow is called the *head*, and the sequence of literals to the right is called the *body*. Syntactically sugared abbreviations, based on this syntax, are defined in the text.

Syntax. We start with a few technical preliminaries. The notation $\vec{X}$ is used to denote a (possibly empty) sequence of items $X_1, ..., X_n$. The syntax of DYNPAL is based on a first order function-free signature $\Sigma = (\mathbf{Const}, \mathbf{Pred})$ with (possibly infinitely many) constants **Const**, and a finite set of predicate names **Pred**. The latter can be divided into two disjoint sets: *static* ($\mathbf{Pred}^{st}$) and *dynamic* ($\mathbf{Pred}^{dy}$) predicate names. The static predicate names can be further divided into two disjoint sets: *intensional* ($\mathbf{Pred}^{in}$) and *extensional* ($\mathbf{Pred}^{ex}$) predicate names. This induces the set of atoms (predicate names applied to expressions of appropriate arity), denoted as **Atom**, which can also be divided into static ($\mathbf{Atom}^{st}$) and dynamic ($\mathbf{Atom}^{dy}$) atoms, and the static ones into intensional ($\mathbf{Atom}^{in}$) and extensional atoms ($\mathbf{Atom}^{ex}$) in the obvious way.

Definition 3.1 (Syntax)**.** *Elementary updates*, static and dynamic *literals* and *rules*, and the terms *guard*, *head* and *body* are defined in Fig. 1. A *state* **B** is a set of ground (i.e. variable-free) extensional atoms. A *policy* **P** is a finite set of rules. □

Definition 3.2 (Variables)**.** The *free variables* of a syntactic phrase W are denoted by $\mathbf{fv}(W)$. The definition is standard (all variables are free variables) apart from the case of elementary updates: $\mathbf{fv}(\pm\{p(\vec{x}) : A\}) = \mathbf{fv}(A) \setminus \vec{x}$. (Hence, the variables $\vec{x}$ are implicitly bound by the set builder construction.) A phrase is *ground* if no variables occur in it, and *closed* if it does not have any free variables. □

Informal overview. As in many static authorisation languages, a policy in DYNPAL is a set of declarative Datalog-like rules built from predicates. Predicates are used for a variety of purposes. *Dynamic* predicate names represent actions (or access requests), and dynamic rules define the conditions and state updates associated with actions, hence dynamic rules are also called *action definitions*. For example, Bob's request to read a file may be represented by `read(Bob,Foo)`, and his request to initiate a payment by `init(Bob,Payment42)`. The policy on a system supporting these requests will likely contain action definitions of the form $\mathtt{read}(x,y) \leftarrow \vec{L}$ and $\mathtt{init}(x,y) \leftarrow \vec{L}'$. *Extensional* predicates represent the current state, and are thus facts that may change over time. Updates, i.e. insertions and retractions, only apply to extensional atoms. For example, the extensional atom `initiated(Bob,Payment42)` may represent the fact that Bob has initiated that particular payment in the past. As in Datalog [13], *intensional* predicates are used to define more complex relations over the extensional atoms in the state; these can then be used as conditions inside dynamic rules.

In Datalog, queries are evaluated with respect to a program (a set of rules) and an extensional database (a set of ground atoms). The head of a rule can be deduced if all body literals can be deduced from the logical closure of the program together with the extensional database. In DYNPAL, the policy corresponds to the program, and the state takes the position of the extensional database. The main difference from Datalog is that deducing dynamic body literals has a side effect on the state. Intuitively, $+\{A_1 : A_2\}$ inserts all atoms A_1 for which A_2 currently holds into the current state. Similarly, $-\{A_1 : A_2\}$ retracts atoms from the state. The deduction (or, equivalently, execution) of a dynamic atom A defined by a rule instance $A \leftarrow \vec{L}$ succeeds if all $L \in \vec{L}$ can be deduced from left to right: if L is static, it is recursively deduced from the policy and the intermediate state at that point in time. If, on the other hand, L is dynamic, it is executed, but its effects are not committed until the end; therefore, the effects of executing an action are *atomic*.

DYNPAL also features negation as failure: $\neg A$ holds if A cannot be deduced. Negation as failure is useful, as many dynamic policies are non-monotonic; in conjunction with retractions, it allows conditions that check for the absence of certain facts in the state. Negation can be problematic in the context of a language with recursive rules, as it leads to non-unique minimal models. We restrict policies to be stratified (adapted from stratified logic programs [38], [2]) in order to separate negation from recursion. Henceforth, we use the term *policy* to refer to a stratified set of rules.

Definition 3.3 (Stratification)**.** A set of rules **P** is *stratified* iff there exists a function σ from **Pred** to natural numbers such that, for all rules $p(\vec{t}) \leftarrow \vec{L}$ in **P** and all literals L occurring in $\vec{L}$ (possibly within an elementary update), the following holds:

1) If $L \equiv q(\vec{u})$, then $\sigma(q) \leq \sigma(p)$.
2) If $L \equiv \neg q(\vec{u})$, then $\sigma(q) < \sigma(p)$. □

Abbreviations. The set of language constructs is de-

signed to be minimal. For practical purposes, it is useful to define some syntactically sugared abbreviations. The equality sign (=) can be used as a built-in predicate, as syntactic equality can be defined as a rule. In the body of a rule, we allow negation not just in front of a single atom, but also in front of a possibly existentially quantified conjunction of static atoms. Similarly, the guard A_2 in an update may also be a possibly existentially quantified conjunction of static literals. Each occurrence of such an abbreviation can be expanded at the expense of introducing a new intensional predicate name and a rule. For example, $p(x) \leftarrow \neg\exists y,z\ (q(x,y), r(z))$ is an abbreviation for the two rules $p(x) \leftarrow \neg\alpha(x)$ and $\alpha(x) \leftarrow q(x,y), r(z)$. We write $\pm p(\vec{t})$ as abbreviation for $\pm\{p(\vec{x})\ :\ \vec{x} = \vec{t}\}$. Some of these abbreviations are used in the examples in the next section.

4. Dynamic Policy Idioms

Policy idioms are abstract reusable solutions to recurring problems in building policies, just like design patterns in software engineering. This section illustrates how some common dynamic policy idioms can be expressed in DYNPAL, before its semantics is formally specified in the following section.

Role sessions. In Cassandra [9], role activation and deactivation within dynamic sessions are special actions inserting and retracting corresponding $\mathsf{hasAct}(\mathit{user}, \mathit{role})$ atoms into and from the state. In DYNPAL, such effects can be specified explicitly. The intensional preconditions canAct and canDeact may be defined by other rules.

$$\mathsf{act}(x,r) \leftarrow \mathsf{canAct}(x,r), \neg\mathsf{hasAct}(x,r), +\mathsf{hasAct}(x,r)$$
$$\mathsf{deact}(x,r) \leftarrow \mathsf{canDeact}(x,r), \mathsf{hasAct}(x,r), -\mathsf{hasAct}(x,r)$$

Cascading role revocation. Cassandra also features a special predicate isDeact for specifying automated deactivation of roles triggered by user-induced role deactivation. This feature can also be mimicked explicitly, by replacing the deactivation rule above by the following one, which uses DYNPAL's conditional bulk update feature.

$$\begin{aligned}\mathsf{deact}(x,r) \leftarrow\ & \mathsf{canDeact}(x,r), \mathsf{hasAct}(x,r),\\ & -\{\mathsf{hasAct}(x',r')\ :\ \mathsf{isDeact}(x',r',x,r)\},\\ & -\mathsf{hasAct}(x,r)\end{aligned}$$

For example, with the rule $\mathsf{isDeact}(x,\mathtt{Stu},y,\mathtt{Supvsr}) \leftarrow \mathsf{hasAct}(x,\mathtt{Stu}), \mathsf{hasAct}(y,\mathtt{Supvsr})$, the deactivation of a supervisor role causes all student roles to be deactivated.

Separation of Duty (SoD). SoD policies come in many variants, many of which depend on a notion of state. Here is an example of *dynamic history-based* SoD, based on the informal example from Section 1. The first rule states that managers can initiate payments that have not yet been initiated. (The predicate init is used for the action of initiating the payment, and initiated is used to record successful payment initiations in the state.)

$$\begin{aligned}\mathsf{init}(x,p) \leftarrow\ & \mathsf{isMgr}(x),\ \neg\exists y\ (\mathsf{initiated}(y,p)),\\ & +\mathsf{initiated}(x,p)\end{aligned}$$

Initiated payments can be cancelled if not yet authorised:

$$\begin{aligned}\mathsf{cancel}(x,p) \leftarrow\ & \mathsf{isMgr}(x),\ \mathsf{initiated}(y,p),\\ & \neg\exists z\ (\mathsf{authorised}(z,p)),\\ & -\{\mathsf{initiated}(v,w)\ :\ w = p, \mathsf{initiated}(v,w)\}\end{aligned}$$

Managers can authorise payments not initiated by themselves:

$$\begin{aligned}\mathsf{auth}(x,p) \leftarrow\ & \mathsf{isMgr}(x),\ \neg\exists z\ (\mathsf{authorised}(z,p)),\\ & \mathsf{initiated}(y,p),\ \neg\mathsf{initiated}(x,p),\\ & +\mathsf{authorised}(x,p)\end{aligned}$$

This is also an example of *order-dependent* SoD, because the various steps within the workflow have to be performed in a specified order. The other variants of dynamic SoD categorised by Simon and Zurko [41] can also be expressed.

Appointment. With the action $\mathsf{app}(x,y,r)$, a user x *appoints* user y to be a member of role r. Appointment is also a feature in administrative role-based access control (ARBAC [39]), and is the most frequently used policy idiom in the Cassandra Electronic Health Record (EHR) case study [5].

$$\begin{aligned}\mathsf{app}(x,y,r) \leftarrow\ & \mathsf{canApp}(x,y,r), \neg\exists x'(\mathsf{hasApp}(x',y,r)),\\ & +\mathsf{hasApp}(x,y,r)\end{aligned}$$

The hasApp predicate can then be used as a precondition for activating roles, for example as in the rule $\mathsf{canAct}(x,r) \leftarrow \mathsf{hasApp}(y,x,r)$. Appointments can be revoked with unapp:

$$\begin{aligned}&\mathsf{unapp}(x,y,r) \leftarrow\\ &\quad \mathsf{canUnapp}(x,y,r), \mathsf{hasApp}(x',y,r),\\ &\quad -\{\mathsf{hasApp}(u,v,w)\ :\ v = y, w = r, \mathsf{hasApp}(u,v,w)\}\end{aligned}$$

To transitively revoke appointments made by revoked appointees, we can use the unapp action from above as a module to compose a new action:

$$\begin{aligned}&\mathsf{unappTrans}(x,y,r) \leftarrow\\ &\quad \mathsf{unapp}(x,y,r),\\ &\quad -\{\mathsf{hasApp}(x',y',r')\ :\\ &\qquad r' = r, \mathsf{hasAppTrans}(y,y',r'), \mathsf{hasApp}(x',y',r')\}\end{aligned}$$

The hasAppTrans predicate used by unappTrans is defined to capture all appointments transitively made by a principal:

$$\begin{aligned}\mathsf{hasAppTrans}(x,y,r) \leftarrow\ & \mathsf{hasApp}(x,y,r)\\ \mathsf{hasAppTrans}(x,y,r) \leftarrow\ & \mathsf{hasAppTrans}(x,y',r),\\ & \mathsf{hasApp}(y',y,r)\end{aligned}$$

Sealed envelopes. In the EHR system currently being developed in the UK [36], patients can hide data from clinicians who would otherwise have read access, by putting the item into a digital *sealed envelope*. Several different options for sealed envelope policies have been identified [6]; the following two rules shows a simple policy where the sealed envelope has two parameters, a target principal and target object.

$$\mathsf{canRead}(x,o) \leftarrow \neg\mathsf{sealed}(x,o)$$
$$\mathsf{seal}(x,y,o) \leftarrow \mathsf{canSeal}(x,y,o), +\mathsf{sealed}(y,o)$$

Integrity constraints. Most action definitions have preconditions, i.e., constraints that have to be satisfied prior to performing the state updates. But in some cases, especially when an action is composed of several other actions, it is useful to have a postcondition (or intermediate conditions) on the updates in the rule. In the example below, action 1 first calls actions 2 and 3 which are defined somewhere else, and then checks that everyone who is a manager also is a user (a containment property, requiring nested stratified negation). If the postcondition fails, the entire action aborts with no effect on the state.

$$\mathsf{doAcn1}(x,o) \leftarrow \mathsf{doAcn2}(x,o), \mathsf{doAcn3}(x,o), \neg\mathsf{notOK}()$$
$$\mathsf{notOK}() \leftarrow \mathsf{isMgr}(x), \neg\mathsf{isUsr}(x)$$

5. DYNPAL Semantics and Proof System

At first sight, inserting and retracting a single atom ($\pm A$) in DYNPAL is reminiscent of the 'assert' and 'retract' operations in Prolog. However, the Prolog operations do not provide an adequate semantics for elementary updates in a dynamic authorisation language. All side effects of an action request are required to be atomic: if the deduction process for a request fails midway, all intermediate updates must be rolled back. For this reason, DYNPAL's semantics is based on transaction logic (TR) [12], [11], an extension of first order logic that can handle negation as failure, has a notion of state and supports transactional state updates.

Translating a DYNPAL rule into a TR formula is straightforward: all free variables of the rule are universally quantified at the outer-most level, and each conjunction symbol (',') is replaced by the sequential operator $\otimes$ from TR. This ensures that multiple elementary updates are executed sequentially and from left to right. In TR, there are no predefined elementary updates. Instead, TR is parameterised by a *transition oracle* $\mathbf{trans_P}$, a function from pairs of states to sets of closed elementary updates. To complete the translation of DYNPAL into TR, we have to define an instance of such a transition oracle that specifies the meaning of $\pm\{A_1 : A_2\}$.

Definition 5.1 (Transition oracle)**.** Let $\mathbf{P}^{st}$ be the set of static rules in a policy $\mathbf{P}$. We write $\mathbf{B} \models^{\mathrm{str}}_{\mathbf{P}} L$ iff the ground static literal L is entailed by $\mathbf{P}^{st}$ (a stratified Datalog program) and $\mathbf{B}$ (according to the standard stratified semantics [2]). The *transition oracle* $\mathbf{trans_P}$ is a function from pairs of states to sets of closed elementary updates, defined as the smallest function satisfying the following equations, where $\pm\{A_1 : A_2\}$ below is closed:

$$+\{A_1 : A_2\} \in \mathbf{trans_P}(\mathbf{B},\mathbf{B}') \text{ iff } \mathbf{B}' = \mathbf{B} \cup \{A_1 : \mathbf{B} \models^{\mathrm{str}}_{\mathbf{P}} A_2\}$$
$$-\{A_1 : A_2\} \in \mathbf{trans_P}(\mathbf{B},\mathbf{B}') \text{ iff } \mathbf{B}' = \mathbf{B} \setminus \{A_1 : \mathbf{B} \models^{\mathrm{str}}_{\mathbf{P}} A_2\}$$

The embedding into TR provides DYNPAL with a model theory, an operational semantics, and various options for extending the language. For lack of space, we do not describe the model theory (for details, see [11]).

Instead, we present a proof system for DYNPAL in Fig. 2 that, for the purpose of this paper, is simpler to use. It defines judgements of the form $\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_n \vdash \vec{L}$, expressing the fact that the sequence of ground literals $\vec{L}$ can be executed in the context of policy $\mathbf{P}$, starting from the initial $\mathbf{B}_0$ and terminating in $\mathbf{B}_n$.

Example 5.2. Consider the SoD example from Section 4. Starting from the initial state $\mathbf{B}_0 = \{\mathsf{isMgr}(\mathtt{A}), \mathsf{isMgr}(\mathtt{B}), \mathsf{initiated}(\mathtt{A},\mathtt{P})\}$, the action $\mathsf{auth}(\mathtt{A},\mathtt{P})$ fails because $\neg\mathsf{initiated}(\mathtt{A},\mathtt{P})$ does not hold in $\mathbf{B}_0$. However, the sequence of actions $\vec{L} = \mathsf{cancel}(\mathtt{A},\mathtt{P}), \mathsf{init}(\mathtt{B},\mathtt{P}), \mathsf{auth}(\mathtt{A},\mathtt{P})$ succeeds. By the proof rules, $\mathbf{P}, \mathbf{B}_0, \mathbf{B}_1, \mathbf{B}_2, \mathbf{B}_3 \vdash \vec{L}$, where $\mathbf{B}_1 = \mathbf{B}_0 \setminus \{\mathsf{initiated}(\mathtt{A},\mathtt{P})\}$, $\mathbf{B}_2 = \mathbf{B}_1 \cup \{\mathsf{initiated}(\mathtt{B},\mathtt{P})\}$, and the final state $\mathbf{B}_3 = \mathbf{B}_2 \cup \{\mathsf{authorised}(\mathtt{A},\mathtt{P})\}$. □

Deterministic access requests. In most authorisation systems, the effects of actions should be finite and deterministic; in other words, only finitely many atoms in the state should be affected by an action, and which atoms are affected should only depend on the action parameters and the current state. To achieve this, we define a syntactic *safety condition* to rule out policies that would cause infinite or non-deterministic effects (e.g. rules such as $a() \leftarrow +q(x)$, which would insert $q(x)$ for all possible instantiations of x; or $a() \leftarrow p(x), +q(x)$, where the effect depends on the choice of how $p(x)$ is proven).

Definition 5.3 (Safety)**.** A static rule is *safe* if the following holds. (*i*) Any variable in the head occurs in a positive atom in the body. (*ii*) Any variable in a negated body atom occurs in a preceding (further to the left) positive atom in the body.

A dynamic rule is *safe* if the following holds. (*iii*) Any free variable in a dynamic literal in the body occurs in the head. (*iv*) Any variable in a negated body atom occurs in the head or in a preceding (further to the left) positive atom in the body. (*v*) For all elementary updates $\pm\{A_1 : A_2\}$ in the body, $\mathbf{fv}(A_1) \subseteq \mathbf{fv}(A_2)$.

A policy $\mathbf{P}$ is *safe* if all its rules are safe, and if for all dynamic rules $p(\vec{t}) \leftarrow \vec{A}$ in $\mathbf{P}$, there is no other rule in $\mathbf{P}$

$$\text{(STA)}\ \frac{\mathbf{B} \models^{\text{str}}_{\mathbf{P}} L \quad L \text{ is ground and static}}{\mathbf{P}, \mathbf{B} \vdash L} \qquad \text{(DYN)}\ \frac{\mathbf{P}, \vec{\mathbf{B}} \vdash \vec{L} \quad A \leftarrow \vec{L} \text{ is a closed instance of a dynamic rule in } \mathbf{P}}{\mathbf{P}, \vec{\mathbf{B}} \vdash A}$$

$$\text{(UPD)}\ \frac{L \in \mathbf{trans_P}(\mathbf{B}_0, \mathbf{B}_1)}{\mathbf{P}, \mathbf{B}_0, \mathbf{B}_1 \vdash L} \qquad \text{(SEQ)}\ \frac{\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_m \vdash L \quad \mathbf{P}, \mathbf{B}_m, ..., \mathbf{B}_n \vdash \vec{L} \quad 0 \leq m \leq n}{\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_n \vdash L, \vec{L}} \qquad \text{(EMP)}\ \frac{}{\mathbf{P}, \mathbf{B} \vdash []}$$

Figure 2. DYNPAL proof system.

with p in its head. □

Requirement (*i*) resembles Datalog's safety condition [13] which ensures a finite number of ground answers for positive static atoms. Together with (*v*), this ensures that each elementary update inserts or retracts only a finite number of ground state atoms. Requirements (*ii*) and (*iv*) ensure that negation is non-floundering, i.e., the negated atom is fully instantiated when it is evaluated (assuming a left-to-right order). Finally, requirement (*iii*), together with the restriction that a policy may hold at most one definition for each dynamic predicate, guarantees that the side effects caused by a successful action request are unique and deterministic.

Proposition 5.4 (Determinism)**.** Let $\mathbf{P}$ be a safe policy and $\mathbf{B}_0$ a finite state. For all $\vec{\mathbf{B}}$, $\vec{\mathbf{B}}'$, and ground $\vec{L}$: if $\mathbf{P}, \mathbf{B}_0, \vec{\mathbf{B}} \vdash \vec{L}$ and $\mathbf{P}, \mathbf{B}_0, \vec{\mathbf{B}}' \vdash \vec{L}$ then $\vec{\mathbf{B}} = \vec{\mathbf{B}}'$, and all states in $\vec{\mathbf{B}}$ are finite.

Proof (sketch). The hard part of the proof is coming up with a stronger proposition that can be proven by induction: for all $\vec{\mathbf{B}}$, $\vec{\mathbf{B}}'$, variable assignments γ and γ' (complete mappings replacing free variables with constants), and literal sequences $\vec{L}$: if $\vec{L}$ has the property that all dynamic literals are ground, and all variables in negated atoms occur in preceding positive static atoms, and if $\mathbf{P}, \mathbf{B}_0, \vec{\mathbf{B}} \vdash \gamma(\vec{L})$ and $\mathbf{P}, \mathbf{B}_0, \vec{\mathbf{B}}' \vdash \gamma'(\vec{L})$, then $\vec{\mathbf{B}} = \vec{\mathbf{B}}'$, and all states in $\vec{\mathbf{B}}$ are finite. This stronger proposition is then proven by standard rule induction. □

6. Reachability and Planning

This section describes a technique for analysing reachability in the context of a DYNPAL policy, by modelling it as an AI planning problem [21], [44].

Understanding and analysing dynamic policies is much harder than static policies, because one has to consider *sequences* of actions of arbitrary lengths, possibly performed by multiple principals. In this context, policy analysis amounts to the problems of *reachability* ('*is there a sequence of actions that leads to a state satisfying the goal property?*') and its dual, *safety* ('*do all sequences of actions lead to a state satisfying the goal property?*'). A wide range of common (and less common) security properties can be phrased in terms of reachability and safety, including effective permission ('*can X eventually do Y?*'), availability ('*can X always do Y?*'), containment ('*are all managers also always users?*') and correspondence queries ('*if a payment has been authorised, has it always been initiated?*').

More formally, a reachability problem consists of a set of constants **Const**, a safe policy $\mathbf{P}$, an initial state $\mathbf{B}_0$ and an extensional formula φ (a first order formula over extensional atoms). A solution to the reachability problem is a sequence of ground actions $\vec{A}$ (with all its constants in **Const**) such that there exist a sequence of states $\mathbf{B}_1, ..., \mathbf{B}_n$ ($n \geq 0$) satisfying

$$\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_n \vdash \vec{A} \text{ and } \mathbf{B}_n \models \varphi.$$

(As usual, the notation $\mathbf{B} \models \varphi$ expresses that the state $\mathbf{B}$ satisfies, i.e. is a (Herbrand) model of, the formula φ.)

In AI, a *plan* is a sequence of operations performed by an agent, or a group of agents, in order to achieve a specified goal. The *planning problem* is then the reasoning task of synthesizing a plan, or else proving that none exists. A planning problem is specified by an *operator description* that defines the preconditions and effects of the supported operations, and a *problem description* that specifies the set of participating agents and objects, the initial state and the goal property. It is not hard to see that the reachability problem in the context of DYNPAL policies naturally corresponds to AI planning, with the policy corresponding to the operator description.

Planning has been at the core of AI since the beginnings, and many varieties, techniques and implementations exist. PDDL [22], a standard language for specifying planning domains, was developed for the International Planning Competition (IPC), which has been held biennially since 1998. By translating the reachability problem into PDDL, we can leverage the search algorithms, heuristics and optimisations of existing PDDL planners.

Operator description. We start by translating a given policy $\mathbf{P}$ into a PDDL operator description. In PDDL, the operator description contains a list of *operator definitions*, each of which consists of an operator name, a list of parameters, a precondition consisting of an extensional formula, and an effect. An effect is a conjunction of elementary effects which include conditional universally quantified insertions and retractions. (For brevity, we do

not use PDDL's concrete syntax here, but note that it is straightforward to model an elementary update in DYNPAL as an elementary effect in PDDL; therefore, for the remainder of the paper, we pretend that the notation $\pm\{A_1 : A_2\}$ is part of PDDL's syntax for effects.) A parameterised operator can be executed if the appropriately instantiated precondition holds in the current state (which, as in DYNPAL, is a set of extensional atoms). If the precondition has been tested positively, the insertions are applied to the current state first, followed by the retractions.

Each dynamic rule in DYNPAL is translated into exactly one equivalent PDDL operator definition. Without loss of generality, we assume that in each dynamic rule in the policy **P**, the head parameters are a sequence of distinct variables. The head of a rule then corresponds to the operator name and the list of parameters. The rest of this section deals with computing the precondition and the effects of the operator definition. The translation process is not entirely straightforward for four reasons, to be dealt with in turn:

1) Intensional and dynamic body literals in DYNPAL rules may be defined by other rules in the policy. In contrast, PDDL operators only allow extensional preconditions and only elementary effects.
2) Static and dynamic body literals may occur in any mixed order in DYNPAL rules. For example, it is possible to express postconditions (or intermediate conditions), as in the body $+p(0), p(0)$ (which will trivially always succeed). PDDL only allows preconditions.
3) All literals within an action execution in DYNPAL are interpreted with respect to intermediate current states that may have been modified by preceding updates. In contrast, conditional effects in PDDL are interpreted with respect to the original state prior to executing the operator. For example, in DYNPAL, execution of the body $+\{p(x) : q(x)\}, -\{p(x) : p(x)\}$ in the state $\{q(0)\}$ results in the same state. In PDDL, the final state would be $\{q(0), p(0)\}$.
4) In DYNPAL, insertions and retractions are executed sequentially, from left to right; in PDDL, retractions are always applied after the insertions. For example, after executing $-p(0), +p(0)$, the atom $p(0)$ would be in the state in DYNPAL, but not in PDDL.

(1) Unfolding. To deal with the first problem, we have to restrict policies to *tight* policies that can be finitely *unfolded* into an extensional formula.

Definition 6.1 (Tightness)**.** A policy **P** is *tight* iff there exists a function τ from ground atoms to natural numbers, such that, for all closed instances (i.e., instances without free variables) $P \leftarrow \vec{L}$ of rules in **P**, the following holds:

1) If $A \in \vec{L}$ is a positive atom, then $\tau(A) < \tau(P)$.
2) If $L = \pm\{A_1 : A_2\}$, then for all ground instances A of A_2, $\tau(A) < \tau(P)$. □

Policy tightness is adapted from the definition of tightness for standard logic programs [34] and enables intensional literals and dynamic atoms to be finitely eliminated by a semantics-preserving transformation called unfolding, also adapted from logic programming [43].

Definition 6.2. Given a safe and tight policy **P**, the *unfolding* of a literal L is an extensional formula defined inductively:

$$\mathbf{unf^P}(A^{ex}) \triangleq A^{ex}, \text{ where } A^{ex} \in \mathbf{Atom}^{ex}$$
$$\mathbf{unf^P}(\neg A) \triangleq \neg\mathbf{unf^P}(A)$$
$$\mathbf{unf^P}(\pm\{A_1 : A_2\}) \triangleq \pm\{A_1 : \mathbf{unf^P}(A_2)\}$$
$$\mathbf{unf^P}(A) \triangleq \bigvee \exists_{\mathbf{fv}(L_1,\ldots,L_k)\setminus\mathbf{fv}(A)} (\mathbf{unf^P}(L_1), \ldots, \mathbf{unf^P}(L_k))$$

The last equation holds for intensional or dynamic atoms A. The big disjunction ranges over all $A' \leftarrow L'_1, \ldots, L'_k$ in **P** such that a most general unifier μ of A and A' exists and $L_i = \mu(L'_1)$, for $i \in \{1, \ldots k\}$. □

(2&3) Interpreting wrt original state. To deal with the second and third problems, we first consider with respect to which state a body literal is interpreted. Consider the execution of a ground action A_g, governed by an action definition with head A, in the context of a safe and tight policy **P**. The unfolding of A yields a (possibly empty) sequence of elementary updates 'padded' by static formulae, or more precisely,

$$\mathbf{unf^P}(A) = \exists\vec{v}\,(\varphi_0, U_0, \ldots, \varphi_{n-1}, U_{n-1}, \varphi_n)$$

for some variables $\vec{v}$ and elementary updates U_i. The φ_i and the guards in the U_i are (possibly empty) extensional formulae. Safety of **P** (in particular the requirement of having at most one rule per dynamic predicate name) ensures that the outermost level of the unfolding is not a disjunction.

Let $\mathbf{B}_0$ be the state before executing the action A_g. If the action can be successfully executed, there is a variable assignment γ (a complete mapping replacing free variables with constants) such that $A_g = \gamma(A)$ and

$$\gamma(\varphi_0, U_0, \ldots, \varphi_{n-1}, U_{n-1}, \varphi_n)$$

can be successfully executed, starting from $\mathbf{B}_0$. According to DYNPAL's semantics, $\gamma(\varphi_0)$ and $\gamma(U_0)$'s guard are interpreted with respect to $\mathbf{B}_0$; $\gamma(\varphi_1)$ and $\gamma(U_1)$'s guard with respect to the state $\mathbf{B}_1$ obtained by applying the update $\gamma(U_1)$ to $\mathbf{B}_0$, and so on. Finally, $\gamma(\varphi_n)$ is interpreted with respect to the final state $\mathbf{B}_n$, obtained by applying $\gamma(U_{n-1})$ to $\mathbf{B}_{n-1}$.

Recall that in PDDL, there are only preconditions and no postconditions or intermediate conditions; this is equivalent to requiring that all extensional formulae be

interpreted with respect to the original state $\mathbf{B}_0$. Similarly, the guards in the conditional effects are also interpreted with respect to $\mathbf{B}_0$. The following transformation step constructs, given $\mathbf{B}_{i+1}$ and an extensional formula φ, a formula φ' such that $\mathbf{B}_{i+1} \vdash \gamma(\varphi)$ (corresponding to DYNPAL semantics) is equivalent to $\mathbf{B}_0 \vdash \gamma(\varphi')$ (corresponding to PDDL semantics). Effectively, it transforms intermediate conditions and postconditions into preconditions.

Proposition 6.3. Let $\gamma, \mathbf{B}_0, ..., \mathbf{B}_n, U_0, ..., U_{n-1}$ be characterised as above, and let φ be an extensional formula. Then for all $i \in \{0, ..., n-1\}$, $\mathbf{B}_{i+1} \vdash \gamma(\varphi)$ iff $\mathbf{B}_i \vdash \gamma(\mathbf{prev}_{U_i}(\varphi))$, where $\mathbf{prev}_{U_i}(\varphi)$ is a defined as follows.

If U_i is of the form $+\{p(\vec{x}) : \psi\}$, then $\mathbf{prev}_{U_i}(\varphi)$ is obtained from φ by replacing all atoms of the form $p(\vec{t})$ by $p(\vec{t}) \vee \psi\{\vec{x} \mapsto \vec{t}\}$. Otherwise, U_i is of the form $-\{p(\vec{x}) : \psi\}$, and $\mathbf{prev}_{U_i}(\varphi)$ is obtained from φ by replacing all atoms of the form $p(\vec{t})$ by $p(\vec{t}) \wedge \neg\psi\{\vec{x} \mapsto \vec{t}\}$.

Proof (sketch). By induction on the structure of φ. The interesting case is the base case $\varphi \equiv p(\vec{t})$. Suppose U_i is of the form $+\{p(\vec{x}) : \psi\}$. Let θ be the partial variable substitution that coincides with γ everywhere except for $\vec{x}$ for which it is undefined. By the definition of $\mathbf{trans_P}$, $\mathbf{B}_{i+1} = \mathbf{B}_i \cup \mathbf{B}'$, where $\mathbf{B}' = \{p(\vec{x}) : \mathbf{B}_i \vdash \theta(\psi)\}$. Hence $\mathbf{B}_{i+1} \models p(\gamma(\vec{t}))$ iff $\mathbf{B}_i \models p(\gamma(\vec{t}))$ or $p(\gamma(\vec{t})) \in \mathbf{B}'$. This set membership is equivalent to $\mathbf{B}_i \models (\theta(\psi))\{\vec{x} \mapsto \gamma(\vec{t})\}$. Since θ does not affect $\vec{x}$ and γ is a total mapping, $(\theta(\psi))\{\vec{x} \mapsto \gamma(\vec{t})\} = \gamma(\psi\{\vec{x} \mapsto \vec{t}\})$. Hence, $\mathbf{B}_{i+1} \models p(\gamma(\vec{t}))$ is equivalent to $\mathbf{B}_i \models \gamma(p(\vec{t}) \vee \psi\{\vec{x} \mapsto \vec{t}\})$, as required. The case for U_i being a retraction is similar. □

A corollary of Proposition 6.3 is the fact that $\mathbf{B}_{i+1} \vdash \gamma(\varphi)$ iff $\mathbf{B}_0 \vdash \gamma\big(\mathbf{prev}_{U_0}(...\mathbf{prev}_{U_i}(\varphi)...)\big)$. Exploiting this fact, we obtain a new formula

$$\exists \vec{v}\,(\varphi'_0, U'_0, ..., \varphi'_{n-1}, U'_{n-1}, \varphi'_n)$$

where $\varphi'_0 = \varphi_0$ and $\varphi'_{i+1} = \mathbf{prev}_{U_0}(...\mathbf{prev}_{U_i}(\varphi_{i+1})...)$, for $i \in \{0, ..., n-1\}$. Similarly, $U'_0 = U_0$ and U'_{i+1} is obtained from U_{i+1} by replacing the guard A_{i+1} by $\mathbf{prev}_{U_0}(...\mathbf{prev}_{U_i}(A_{i+1})...)$.

This formula takes into account that literals in PDDL are interpreted with respect to $\mathbf{B}_0$, as opposed to the state obtained by applying all intermediate updates further to the left. Hence we can safely reorder the literals to obtain

$$\exists \vec{v}\,(\varphi'_0, ..., \varphi'_n,\ U'_0, ..., U'_{n-1}).$$

(4) Insertions before retractions. Finally, to deal with the fourth problem, we show how a retraction followed by an insertion can be converted into an equivalent insertion-retraction pair. By repeatedly applying this operation to the formula obtained above, the sequence $U'_0, ..., U'_{n-1}$ can be transformed into an equivalent one in which all insertions happen before the retractions.

If the retraction and the insertion affect different predicate names, there is no interference between them, so they can be swapped without modification. Now we consider the case where there *is* interference between the two updates.

Proposition 6.4. Let $U_- = -\{p(\vec{x}) : \psi_-\}$ and $U_+ = +\{p(\vec{x}) : \psi_+\}$ (without loss of generality, we assume that the same parameter $\vec{x}$ is used for p in both updates). Let $U'_- = -\{p(\vec{x}) : \psi_- \wedge \neg\psi_+\}$. Then applying U_- before U_+ has the same effect as applying U_+ before U'_- (where the guards are interpreted with respect to the same state).

Proof (sketch). This follows from the definition of $\mathbf{trans_P}$ (modified to interpret the guard with respect to a given state) and the set identity $A \setminus B \cup C = A \cup C \setminus (B \setminus C)$. □

The swapping operation from Proposition 6.4 is applied repeatedly to yield the formula

$$\exists \vec{v}\,(\varphi'_0, ..., \varphi'_n,\ U''_0, ..., U''_{n-1}),$$

which has the property that all insertions are to the left of the retractions. By safety of $\mathbf{P}$ and by the definition of $\mathbf{unf^P}$, no variable in $\vec{v}$ occurs as a free variable in any of the elementary updates; therefore, the scope of the outermost existential quantifier can be restricted to the extensional formulae φ'_i.

The resulting PDDL operator definition for A has the *precondition* $\exists \vec{v}\,(\varphi'_0, ..., \varphi'_n)$ and the conjunctive *effect* $U''_0, ..., U''_{n-1}$.

Problem description. Recall that a planning problem consists of an operator description and a problem description. The method described above provides a translation of a safe and tight policy $\mathbf{P}$ into a PDDL operator description. A PDDL *problem description* consists of a finite enumeration of constants **Const** (e.g. principals, resources, parameters) that may be used in a plan, a specification of the initial state (a list of of ground atoms that are initially true), and a goal property φ (any extensional formula).

For example, to check X's effective permission for executing the ground action A, we compute the PDDL preconditions of A using the translation process described above and use the resulting formula as goal property in the problem description. Any planner supporting PDDL can then be used to search for a plan.

Complexity and decidability. The translation to AI planning also gives us complexity and decidability results (due to Erol et al. [20]).

Proposition 6.5. Given a set of constants **Const**, a safe and tight policy $\mathbf{P}$, a state $\mathbf{B}_0$ and an existentially closed conjunction of extensional atoms φ, deciding whether there exist $\mathbf{B}_1, ..., \mathbf{B}_n$ and $\vec{A}$ (with constants drawn from **Const**),

such that

$$\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_n \vdash \vec{A} \text{ and } \mathbf{B}_n \models \varphi,$$

is strictly semi-decidable for infinite **Const**, and EXPSPACE-complete for finite **Const** (where the input is **P**, $\mathbf{B}_0$ and φ). If **P** is a fixed parameter and **Const** is finite, and all operator definitions in the PDDL translation of **P** have purely conjunctive preconditions, the problem is in PSPACE. It is in P if, additionally, there are no negative preconditions and no retractions. If, additionally, each operator definition contains at most one precondition, the problem is in NLOGSPACE.

7. Invariants and Theorem Proving

This section describes a second method for analysing DYNPAL policies that complements the planning method from Section 6. The planning method provides a solution to the reachability problem: if the answer is positive, a sequence of actions that leads to a goal state is produced in the form of a plan. The dual of the reachability problem, *safety* ('*do all reachable states satisfy the safety property?*'), can also, in principle, be solved by the planning method: we just check that no plan exists that leads to a 'bad' state.

However, the planning method is of limited use for verifying safety in practice, because (i) it requires the initial state and the domain over which variables range to be specified, and (ii) the specified domain has to be finite. (Methods based on model checking, e.g. [16], [45], have the same limitations.) Thus if the answer returned by the planner is negative, it is only guaranteed that there exists no plan starting from the *given* state and using the *given* finite list of constants. It does not rule out the possibility that there may be a plan leading to an unsafe state involving more constants, for instance for a sequence of actions performed by a larger coalition of possibly unknown principals. This limitation can be problematic in decentralised systems, where the principals interacting with each other typically are mutual strangers at first, so their identities are not known a priori, and their number is unbounded. From Proposition 6.5, we know that checking non-reachability in infinite domains is undecidable.

Moreover, even though modern planners are equipped with a library of heuristics to quickly find plans or to prove that no plan exists, there are still situations where an exhaustive search is necessary, which is only feasible for relatively small sets of constants. (Again, finite model checking suffers from the same problem.)

The method presented in this section checks if a property, formulated in first order logic (FOL), is an *invariant* of a given dynamic policy, i.e., a safety property that is preserved over any sequence of actions, with parameters drawn from *any* arbitrary domain, including *infinite* ones. This is formalised below.

Definition 7.1. An *invariant* φ of a policy **P** is an extensional formula such that for all $\mathbf{B}_0, ..., \mathbf{B}_n, \vec{A}$ the following holds: if $\mathbf{B}_0 \models \varphi$ and $\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_n \vdash \vec{A}$ then $\mathbf{B}_n \models \varphi$. □

Note that $\vec{A}$ is a sequence of (possibly dynamic) *atoms*, not literals; in particular, it does not contain any elementary updates. Therefore, the invariance only has to hold for sequences of actions defined in the policy, not for any arbitrary updates.

The remainder of this section describes how a FOL formula is constructed from an invariant candidate and a policy. We then prove that this formula is (finitely) valid if and only if the candidate is indeed an invariant. The invariance problem can thus be solved by any general-purpose theorem prover that can prove validity and generate finite counter examples.

The execution of an action typically involves a sequence of elementary updates. To model such an execution in FOL, we make the truth values of extensional atoms before and after each intermediate update explicit. Intuitively, each intermediate update within an execution is taken as a discrete time step, so predicate names are tagged with time steps to keep track of state changes. For example, p_0 relates to predicates p at time step 0, p_1 is used for p-atoms one update later, and so on. Formally, let $\mathbf{Pred}_\mathbb{N}$ be a countably infinite set of predicate names disjoint from **Pred**. We define an injective function $\mathbf{step} : \mathbf{Pred} \times \mathbb{N} \to \mathbf{Pred}_\mathbb{N}$, and write p_n as shorthand for $\mathbf{step}_n(p)$. The function **step** is extended to formulae φ over **Pred** such that $\mathbf{step}_n(\varphi)$ denotes the same formula with p replaced by p_n, for all predicate names p. Similarly, $\mathbf{step}_n(\mathbf{B})$ denotes the state obtained from **B** by replacing all predicate names p by p_n. We define $\mathbf{step}^{-1} : \mathbf{Pred}_\mathbb{N} \to \mathbb{N}$ such that $\mathbf{step}^{-1}(p_n) = n$. This function is extended to formulae φ over signature $(\mathbf{Const}, \mathbf{Pred}_\mathbb{N})$ such that $\mathbf{step}^{-1}(\varphi) = \mathbf{max}\{\mathbf{step}^{-1}(p) : p \text{ occurs in } \varphi\}$. The construction requires a couple of auxiliary definitions.

Definition 7.2. If φ, ψ_1, ψ_2 are formulae, **if** φ **then** ψ_1 **else** ψ_2 is short for the formula $(\varphi \to \psi_1) \wedge (\neg\varphi \to \psi_2)$. □

Definition 7.3. For all $p \in \mathbf{Pred}^{ex}$, let $\mathbf{frame}^n_p$ denote

$$\bigwedge_{q \in \mathbf{Pred}^{ex}, q \neq p} \forall \vec{x}(q_n(\vec{x}) \leftrightarrow q_{n+1}(\vec{x})),$$

where the $\vec{x}$ are distinct and have appropriate arities (depending on the q's). □

The formula $\mathbf{frame}^n_p$ from Definition 7.3 is used in the translation to express the fact that, at time step $n+1$, the truth values of all atoms remain as they were at step n, apart from p-atoms. It can be seen as a naive, but for our

$$
\begin{array}{lcll}
\mathbf{fol}^{\mathbf{P}}_n(p(\vec{t})) & \triangleq & p_n(\vec{t}) & \text{if } p \in \mathbf{Pred}^{ex} \\
\mathbf{fol}^{\mathbf{P}}_n(\neg A) & \triangleq & \neg \mathbf{fol}^{\mathbf{P}}_n(A) & \\
\mathbf{fol}^{\mathbf{P}}_n(+\{p(\vec{x}) : A\}) & \triangleq & \mathbf{frame}^n_p \wedge \forall \vec{x} \left(\textbf{if } \mathbf{fol}^{\mathbf{P}}_n(A) \textbf{ then } p_{n+1}(\vec{x}) \textbf{ else } p_n(\vec{x}) \leftrightarrow p_{n+1}(\vec{x}) \right) & \\
\mathbf{fol}^{\mathbf{P}}_n(-\{p(\vec{x}) : A\}) & \triangleq & \mathbf{frame}^n_p \wedge \forall \vec{x} \left(\textbf{if } \mathbf{fol}^{\mathbf{P}}_n(A) \textbf{ then } \neg p_{n+1}(\vec{x}) \textbf{ else } p_n(\vec{x}) \leftrightarrow p_{n+1}(\vec{x}) \right) & \\
\mathbf{fol}^{\mathbf{P}}_n(A) & \triangleq & \bigvee \exists_{\mathbf{fv}(L_1,\ldots,L_k)\setminus\mathbf{fv}(A)} \left(\mathbf{fol}^{\mathbf{P}}_{r_1}(L_1) \wedge \ldots \wedge \mathbf{fol}^{\mathbf{P}}_{r_k}(L_k) \right) &
\end{array}
$$

Figure 3. Translating a safe and tight policy $\mathbf{P}$ into FOL. The big disjunction in the last equation ranges over all rules $A' \leftarrow L'_1,\ldots,L'_k$ in $\mathbf{P}$ such that a most general unifier μ of A and A' exists and $L_i = \mu(L'_1)$, for $i \in \{1,\ldots k\}$. Furthermore, $r_1 = n$, and $r_i = \mathbf{step}^{-1}(\mathbf{fol}^{\mathbf{P}}_{r_{i-1}}(L_{i-1}))$ for $i \in \{2,\ldots,k\}$.

purposes feasible way of addressing the frame problem [35].

Based on these auxiliary definitions, we can now define the centrepiece of the invariance formula: given a natural number n, a literal L, and a safe and tight policy $\mathbf{P}$, $\mathbf{fol}^{\mathbf{P}}_n(L)$ is inductively defined in Fig. 3. The first case states that if the extensional literal $p(\vec{t})$ can be executed successfully at step n, then it must be true at that step, represented by $p_n(\vec{t})$. The cases for elementary updates state that if the guard A is true at step n then the affected atom $p(\vec{x})$ is true at step $n+1$ (or false, in the case of retraction); furthermore, all other atoms remain unaffected. The second and the last case eliminate intensional and dynamic atoms by unfolding, as in the previous section, but with time-step tagging. Note that the definition is well-founded because $\mathbf{P}$ is tight, and hence the unfolding operation of the last case can only be applied a finite number of times.

Intuitively, $\mathbf{fol}^{\mathbf{P}}_n(L)$ models all possible successful executions of L by characterising the truth relationships between all extensional atoms at each intermediate step, assuming that the execution starts at step n. It expresses the strongest postcondition on states for each intermediate step, irrespective of the concrete initial state or the domain. It could thus be seen as an abstract representation of the executions of a literal. The following lemma formalises this intuition.

Lemma 7.4. For all m, safe and tight policies $\mathbf{P}$, ground L, and $\mathbf{B}_m$,$\mathbf{B}_{m+1}$,...,$\mathbf{B}_n$, where $n = \mathbf{step}^{-1}(\mathbf{fol}^{\mathbf{P}}_m(L))$, the following holds:

$$\mathbf{P}, \mathbf{B}_m, \ldots, \mathbf{B}_n \vdash L \text{ iff } \bigcup_{i=m}^{n} \mathbf{step}_i(\mathbf{B}_i) \models \mathbf{fol}^{\mathbf{P}}_m(L)$$

Proof (sketch). By rule induction on the proof system. The different cases in the definition of $\mathbf{fol}^{\mathbf{P}}$ correspond directly to the proof rules. In particular, the case definitions for elementary updates are direct encodings of $\mathbf{trans}_{\mathbf{P}}$ in first order logic: for any elementary update L, we have $L \in \mathbf{trans}_{\mathbf{P}}(\mathbf{B}_m, \mathbf{B}_{m+1})$ iff $\mathbf{step}_m(\mathbf{B}_m) \cup \mathbf{step}_{m+1}(\mathbf{B}_{m+1}) \models \mathbf{fol}^{\mathbf{P}}_m(L)$. □

Two further auxiliary definitions are needed for the final step of generating the invariance formula:

Definition 7.5. A formula φ is *finitely valid*, written $\models^{fin} \varphi$, if every finite interpretation (i.e., a structure with finite domain) is a model of φ. (Or equivalently, if $\neg\varphi$ has no finite model.)

Definition 7.6. Given a policy $\mathbf{P}$, let $\mathbf{una}^{\mathbf{P}}$ denote the formula $\bigwedge \neg(c_1 = c_2)$, where the conjunction ranges over all pairs of constants $c_1 \neq c_2$ that occur in $\mathbf{P}$. □

The following lemma shows that a property φ on the *initial* state, together with $\mathbf{fol}^{\mathbf{P}}_0(L)$ and the unique name assumption $\mathbf{una}^{\mathbf{P}}$ (distinct constant symbols are interpreted as distinct constants), logically implies a property ψ on the *final* state (under all finite interpretations) if and only if for all concrete executions of L starting from any initial state satisfying φ, the execution terminates in a final state satisfying ψ. The restriction of *finite* validity and the unique name assumption are essential for proving the lemma, as only finite Herbrand models of the implication are guaranteed to correspond to real executions (where all intermediate states are finite).

Lemma 7.7. For all extensional formulae φ, ψ, safe and tight policies $\mathbf{P}$, L, and n, where L is ground and $n = \mathbf{step}^{-1}(\mathbf{fol}^{\mathbf{P}}_0(L))$, the following holds: $\models^{fin} \mathbf{step}_0(\varphi) \wedge \mathbf{fol}^{\mathbf{P}}_0(L) \wedge \mathbf{una}^{\mathbf{P}} \Rightarrow \mathbf{step}_n(\psi)$ iff for all $\mathbf{B}_0$,...,$\mathbf{B}_n$,

$$\mathbf{B}_0 \models \varphi \text{ and } \mathbf{P}, \mathbf{B}_0, \ldots, \mathbf{B}_n \vdash L \text{ imply } \mathbf{B}_n \models \psi.$$

Proof. For the 'only if' direction, Lemma 7.4, applied to the assumption $\mathbf{P}, \mathbf{B}_0, \ldots, \mathbf{B}_n \vdash L$, implies $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i) \models \mathbf{fol}^{\mathbf{P}}_m(L)$. Secondly, from the assumption $\mathbf{B}_0 \models \varphi$, we can deduce $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i) \models \mathbf{step}_0(\varphi)$. Thirdly, since $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i)$ is a Herbrand interpretation, it entails $\mathbf{una}^{\mathbf{P}}$. Fourthly, note that $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i)$ is a finite interpretation. Combining these four facts with the main assumption gives $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i) \models \mathbf{step}_n(\psi)$. Since $\mathbf{step}$ is injective, it follows that $\mathbf{step}_n(\mathbf{B}_n) \models \mathbf{step}_n(\psi)$, which implies $\mathbf{B}_n \models \psi$.

For the 'if' direction, consider any finite model of $\mathbf{step}_0(\varphi) \wedge \mathbf{fol}^{\mathbf{P}}_0(L) \wedge \mathbf{una}^{\mathbf{P}}$. This model has a submodel that is isomorphic to a Herbrand model of the form $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i)$ where each $\mathbf{B}_i$ is finite. We need to show

that this model satisfies $\mathbf{step}_n(\psi)$. Since $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i)$ satisfies $\mathbf{step}_0(\varphi)$, so does $\mathbf{step}_0(\mathbf{B}_0)$, by injectivity of **step**, and hence $\mathbf{B}_0 \models \varphi$. Furthermore, by Lemma 7.4, $\mathbf{P}, \mathbf{B}_0, ..., \mathbf{B}_n \vdash L$. Thus the implication of the main assumption gives $\mathbf{B}_n \models \psi$, and equivalently $\mathbf{step}_n(\mathbf{B}_n)$ is a model of $\mathbf{step}_n(\psi)$. By injectivity of **step**, this model can be padded with 'irrelevant' atoms, thus $\bigcup_{i=0}^{n} \mathbf{step}_i(\mathbf{B}_i) \models \mathbf{step}_n(\psi)$. □

Finally, Theorem 7.8 constructs the complete formula for checking invariants:

Theorem 7.8. For all extensional formulae φ and safe and tight policies **P** the following holds: φ is an invariant of **P** iff for all $p \in \mathbf{Pred}^{dy}$,

$$\models^{fin} \forall \vec{x} : \mathbf{step}_0(\varphi) \wedge \mathbf{fol}_0^{\mathbf{P}}(p(\vec{x})) \wedge \mathbf{una}^{\mathbf{P}} \Rightarrow \mathbf{step}_n(\varphi),$$

where $n = \mathbf{step}^{-1}(\mathbf{fol}_0^{\mathbf{P}}(p(\vec{x})))$ (and $\vec{x}$ has the arity of p).

Proof. The 'if' direction is proven by induction on the length of $\vec{A}$ (in Definition 7.1). For sequences of length 1, the invariance property is trivially satisfied for non-dynamic atoms. For dynamic atoms, the statement follows from the 'only if' part of Lemma 7.7 (with $\varphi = \psi$). For sequences of greater lengths, the statement follows from the inductive hypothesis and the (SEQ) rule. The 'only if' direction is a corollary of the 'if' part of Lemma 7.7 (again with $\varphi = \psi$). □

The theorem establishes soundness and completeness of the algorithm and justifies the use of an automated FOL theorem prover for proving and disproving invariants. However, it has to be a theorem prover that can not only prove validity, but, in the case of non-validity, can also prove the formula is not *finitely* valid. For instance, the combination [37] of Prover9 (an automated theorem prover) and Mace4 (a finite model searcher for FOL) is suitable for this purpose: Prover9 and Mace4 are typically run in parallel; if the formula from Theorem 7.8 is proven valid by Prover9, it follows that it is also finitely valid, therefore φ is an invariant. On the other hand, if Mace4 reports that the formula is not valid, this implies that a *finite* counter example exists, hence the formula is not finitely valid, and so φ is provably *not* an invariant. The reachability analysis from Section 6 could then be used to find a concrete sequence of actions that violates the property.

8. Discussion

Experimental results. The case studies in Zhang et al. [45] with the RW framework were used for obtaining some preliminary experimental results. In particular, the authorisation policies for their conference paper review system (CRS), their employee information system (EIS) for managing bonus allocations, and their student information system (SIS) for managing students' marks were encoded in DYNPAL. (For lack of space, these are not described here.) The encoding was straightforward and in some cases simpler than the RW version due to DYNPAL's support of intensional predicates.

The relevant reachability queries from their paper were then evaluated, using the AI planner FF [26] and the theorem prover Prover9-Mace4 [37]. Only very brief descriptions of the queries are provided here (for details, see [45]): Query 4.2 aims to find a strategy for a conference chair to promote an agent to be a reviewer of a paper. Query 4.3 attempts to find a way for an agent to read a review before having submitted a review for that paper herself. Query 6.4 asks if two managers can cooperate to set each other's bonuses. Query 6.8 tries to find a strategy where a lecturer appoints two students to be demonstrators of each other (thereby enabling them to set each other's marks).

The computation times alongside the RW results based on model checking with AcPeg [45] are reported in Fig. 4. Note that the respective computation times cannot be directly compared, as Zhang et al.'s notion of reachability takes into account the *knowledge state* of principals: a goal is only considered to be reached if the principals *know* (by performing read actions) that it has been reached. In the presence of incomplete knowledge, reading a state variable may return $\top$ *or* $\bot$ (their variables only range over a boolean domain). It is likely that this nondeterminism causes their reachability queries to be more complex. Nevertheless, the numbers in Fig. 4 show that the model checking approach is very sensitive to the number of agents and other constants in the environment, as the resulting model grows exponentially with the size of the domain. In contrast, the goal-directed heuristics of the planner tend to find a plan very quickly, irrespective of the size of the domain. This coincides with the intuition that model checking is better at proving that no plan exists (but only for finite domains), whereas AI planning is optimised for finding plans (although there is a growing body of work seeking to marry the two approaches [30], [23], [18]).

Most plans found were short (≤ 3 steps); to construct a problem with a longer plan, Query 4.3 was modified to start from a state containing only a conference chair person. The resulting plan consisted of 8 steps, and was computed in 140 ms. All queries have a positive answer apart from Query 6.8. Here, the planner failed to prove that no plan exists within reasonable time, but proving the invariant that no two students can be demonstrators of each other (even with infinitely many agents) only took 1,114 ms (234 ms to generate the invariance formula, and 880 ms to run the theorem prover). (To be precise, the invariant also contained the provision that the relation stating that one

Query (from [45])	Constants	DYNPAL	RW
Query 4.2	7	120	3,600
Query 4.3	4	125	250
Query 6.4	6	120	288
Query 6.4	12	120	22,500
Query 6.4	18	120	177,357
Query 6.8	10	∞/1,114	33,807

Figure 4. Query evaluation times in ms. Caveat: this does not serve as a direct comparison, as the notion of reachability in RW differs somewhat from this paper's (see text).

student is in a higher year than another is anti-symmetric.) Other interesting invariants were also proven in the context of the CRS, for example that anyone who is reviewing or subreviewing a paper or has submitted a review for a paper, is not an author of that paper. Again, the result holds for any number of papers and agents in the system. Generating this invariance formula took 990 ms, proving finite validity 3,200 ms.

To demonstrate how DYNPAL's dynamic features can be added to existing languages, SecPAL [7] was extended by a fragment of DYNPAL (essentially, the SMP [8] fragment). The state was stored in an SQL database to enable fast lookups and updates. To test the scalability of the system, the nation-wide health record policy from the Cassandra case study [5] was implemented, resulting in 46 rules (of which 23 are action definitions), as well as web-based patient and clinician portals as front ends for reading, writing and annotating health records, referring patients, managing sealed envelopes etc. The state was populated with two million random atoms about records, patients and clinicians. Despite the large size of the state and the complexity of the policy, policy evaluation and update times were insignificantly low.

Limitations and future work. While DYNPAL's dynamic features can be easily added to existing logic-based authorisation languages, the analysis methods and results cannot always be carried over, if the base language is more expressive than Datalog. For example, if the base language is as expressive as Datalog with constraints (as in SecPAL), the planning analysis would still work (since the domain is finite), but the invariance analysis would require theorem proving over formulas with constraints.

One important topic not discussed here is that of correctly and efficiently implementing DYNPAL's transactional execution semantics. In particular, techniques from database transactions and concurrency control (such as STM) may be applicable. This issue is largely orthogonal to the results and discussions in this paper.

DYNPAL is designed to ensure that the effects of actions are deterministic. Barth and Mitchell [4] argue that this is not a reasonable property for digital rights management, where an action (such as playing a song) may consume or modify one out of several possible existing licenses. Such non-deterministic actions can be expressed in DYNPAL by relaxing the safety condition from Definition 5.3. However, analysis would become much harder; indeed, as shown by Barth and Mitchell, even just checking if a *given* sequence of actions is permitted is intractable.

RW [45] is less expressive than DYNPAL, as it does not support postconditions, intermediate conditions, and actions that update more than one fact. However, their framework supports more complex queries than those considered in Section 6. Firstly, RW queries can be nested (e.g. '*can we reach a state in which* φ *holds, and from there reach a state in which* ψ *holds?*'). Secondly, explicit coalitions of agents who may be involved in the actions may be specified (e.g. '*can A and B together reach X?*'). Thirdly, RW distinguishes between strategies and guessing strategies (see Section 2). To support queries with some of these extended features, it may be possible to pre-transform the policy appropriately before the method in Section 6 is applied. The details are beyond the scope of this paper, but may be addressed in a future paper.

Invariance analysis, as described in Section 7, is useful for debugging and verifying policies. However, as in program verification, it is often difficult to come up with correct invariants. Therefore, some form of automated assistance in generating policy invariants would be desirable; this problem may be pursued in future work.

Acknowledgements

I would like to thank Paul Fodor, Michael Kifer, Mark Ryan and Nikhil Swamy for helpful comments and discussions, and Jorge Baier, Jörg Hoffmann and Christoph Weidenbach for their advice on planners and theorem provers. The SecPAL extension and the EHR policy were implemented together with Mahmoud Moadeli.

References

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] A. K. Bandara, E. C. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *4th IEEE International Workshop on Policies for Distributed Systems and Networks*, page 26, 2003.

[4] A. Barth and J. C. Mitchell. Managing digital rights using linear logic. In *21st IEEE Symposium on Logic in Computer Science*, pages 127–136, 2006.

[5] M. Y. Becker. Cassandra: Flexible trust management and its application to electronic health records. Technical Report UCAM-CL-TR-648, University of Cambridge, Computer Laboratory, 2005.

[6] M. Y. Becker. Information governance in NHS's NPfIT: A case for policy specification. *International Journal of Medical Informatics*, 76(5-6), 2007.

[7] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[8] M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. In *12th European Conference on Research in Computer Security (ESORICS)*, volume 4734 of *Lecture Notes in Computer Science*, pages 203–218, 2007.

[9] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 139–154, 2004.

[10] M. Blaze, J. Feigenbaum, and A. D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[11] A. Bonner and M. Kifer. Transaction logic programming. Technical Report Technical Report CSRI-323, Computer Systems Research Institute, University of Toronto, 1995.

[12] A. J. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[13] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[14] S. Chen, D. Wijesekera, and S. Jajodia. Incorporating dynamic constraints in the flexible authorization framework. In *European Symposium on Research Computer Security*, pages 1–16, 2004.

[15] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *International Workshop on Policies for Distributed Systems and Networks*, pages 18–38, 2001.

[16] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006*, pages 632–646, 2006.

[17] D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Obligations and their interaction with programs. In *12th European Symposium On Research In Computer Security (ESORICS)*, volume 4734 of *Lecture Notes in Computer Science*, pages 375–389, 2007.

[18] S. Edelkamp and M. Helmert. MIPS: The model-checking integrated planning system. *AI Magazine*, 22(3):67–72, 2001.

[19] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory, RFC 2693, September 1999.

[20] K. Erol, D. S. Nau, and V. S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

[21] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, pages 189–208, 1971.

[22] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20, 2003.

[23] F. Giunchiglia and P. Traverso. Planning as model checking. In *5th European Conference on Planning*, pages 1–20, 1999.

[24] Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 149–162, 2008.

[25] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[26] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[27] R. Jagadeesan, W. Marrero, C. Pitcher, and V. Saraswat. Timed constraint programming: a declarative approach to usage control. In *International Conference on Principles and Practice of Declarative Programming*, pages 164–175, 2005.

[28] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[29] R. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[30] S. Kupferschmid, J. Hoffmann, H. Dierks, and G. Behrmann. Adapting an AI planning heuristic for directed model checking. In *13th International SPIN Workshop*, volume 3925 of *Lecture Notes in Computer Science*, pages 35–52, 2006.

[31] N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.

[32] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[33] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: security analysis in trust management. *Journal of the ACM*, 52(3):474–514, 2005.

[34] V. Lifschitz. Foundations of logic programming. *Principles of Knowledge Representation*, pages 69–127, 1996.

[35] J. Mccarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502, 1969.

[36] National Health Service, UK. Integrated Care Records Service: Output based specification version 2. 2003.

[37] Prover9-Mace4. Version December 2007, http://www.cs.unm.edu/~mccune/prover9.

[38] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. *Foundations of deductive databases and logic programming*, pages 193–216, 1988.

[39] R. Sandhu, V. Bhamidipati, E. Coyne, S. Canta, and C. Youman. The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In *2nd ACM Workshop on Role-Based Access Control*, pages 41–54, 1997.

[40] A. Sasturkar, P. Yang, S. D. Stoller, and C. R. Ramakrishnan. Policy analysis for administrative role based access control. In *IEEE Computer Security Foundations Workshop*, pages 124–138, 2006.

[41] R. Simon and M. E. Zurko. Separation of duty in role-based environments. In *10th IEEE Computer Security Foundations Workshop (CSFW)*, pages 183–194, 1997.

[42] S. D. Stoller, P. Yang, C. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *14th ACM Conference on Computer and Communications Security (CCS)*. ACM Press, 2007.

[43] H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *International Conference on Logic Programming*, pages 127–138, 1984.

[44] D. S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[45] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.