A Logic Programming Approach to Incorporate Access Control in the Internet of Things

Ilse Bohé, Michiel Willocx, Jorn Lapon, Vincent Naessens

A Logic Programming Approach to Incorporate Access Control in the Internet of Things

Ilse Bohé[1], Michiel Willocx[1], Jorn Lapon[1], and Vincent Naessens[1]

DistriNet - KU Leuven, Technology Campus, Ghent, Belgium
{first.last}@kuleuven.be

Abstract. In the present digital world, we depend on information technology more than ever. Our economy, health, well-being and even our lives depend on it. Information security is a basic requirement, with access control playing a key role in limiting potential risks. However, the digital environment is no longer limited to data, access to the IoT space must also be handled properly.
Logic has shown to be very useful in access control. It has been used to formally explain and verify access control policies. Here, logic is employed as a reasoning service in support of other systems. However, a general access control mechanism for logic programs is not available.
This paper presents a structural approach that brings Access Control to Logic Programming. It allows to constrain access to the knowledge base, supporting the use of impure predicates, preventing unauthorized side effects (i.e. controlling IoT devices) taking place.

Keywords: access control · logic programming · impure logic · IoT

1 Introduction

The technological revolution has given rise to impactful technologies. Sharing and processing information is a key factor for improvement. Unfortunately, the many examples of data breaches [6, 3] and the loss of privacy represent a permanent threat. Several access control strategies have been developed to help in securing this data. Although access control may seem conceptually straightforward, its integration is often complex and error-prone. Over the years, research on access control that harnesses logic is substantial: it has been used to formally verify security properties; to explain, express and enforce access control policies, etc. While knowledge bases may house huge amounts of data and knowledge, research on the use of access control within knowledge representation and reasoning systems is very limited. As former research shows, logic easily lends itself in expressing and enforcing access control policies. However, no structural approach is available to enforce access control inside logic programs.

This work examines how common access control mechanisms can be enforced in logic programs such as programs written in Prolog, Datalog, Logica [9] and Yedalog [5]. A straightforward approach would be to verify access control policies and remove predicates that do not comply with the policy during consultation of

the program. This mimics standard access control to resources as a whole (e.g. a file). In logic programs, however, access control can be much more versatile. Not only access to data or entities can be controlled, but also access to knowledge (i.e. the reasoning itself). Resulting in complex access control logic. Intuitively, when access is denied, the knowledge should appear as nonexistent, and the user only has a limited view on the knowledge base. In other words, queries requiring inaccessible knowledge for its reasoning, do not return results. Otherwise, they must produce the same results as if no access control were used. In that sense, impure predicates require special care. Impure predicates result in side effects, when the predicate is resolved [17]. For instance, consider the `open(Lock)` predicate that opens the smart door lock, `Lock`. It is impossible to revert the side effects upon backtracking. In the light of access control, it is therefore very important that side effects only occur when allowed.

Contributions. This paper presents a solution that evaluates access control policies during resolution in logic programs, taking special care for impure predicates, (e.g., actions in the IoT space). It provides a high expressiveness and fine-grained control of the program and makes it a widely applicable approach. A *deny as soon as possible* strategy is used, but decisions are postponed until they can be decided with certainty. Moreover, as enforcement occurs during inference, the approach easily extends to the dynamic case such as a reactive system (i.e., one that responds to external inputs). In this approach, access rules are defined as part of the program logic. In other words, the rules can take advantage of the program's knowledge base. Hence, expressing access control strategies, such as resource based, role based and relationship based access control, is straightforward. To validate and demonstrate the approach, an implementation is provided as a Prolog meta-interpreter, named ACoP. It can easily be integrated in existing Prolog programs with minimal effort. Overhead is limited to defining the access rules, also in Prolog.

The remainder of this paper is structured as follows. Section 2 points to related work in the field of access control. Section 3 explains the design of the proposed solution. More details on the Prolog implementation of ACoP can be found in Section 4 together with an evaluation of the implementation. Section 5 discusses the work and the paper ends with conclusions.

2 Previous & Related Work

A Logic Programming IoT Reasoning Middleware The work builds on a previously proposed reasoning middleware for the IoT [2]. The middleware hosts a modular, module-based, logic reasoner, developed in Prolog. Each module has its own functionality, including access control. In this work we look at how the existing access control module can be extended to support multiple access control strategies, without increasing the overhead for the developer.

Access Control. For many years, logic programming has been used to support access control [15, 1, 12]. Also more recent work takes advantage of formal logic

to realize and verify access control models. There are several established access control models, ranging from easy to implement strategies, such as consulting an access control matrix, over rule based access control (RBAC), to more complex strategies such as organizational based access control (OrBAC) [18] and relationship based access control (ReBAC) [8]. Huynh et. al defined an alternative strategy that uses priority, modality and specificity to handle conflicts [10]. The multi-layered access control model was implemented in both ProB and Alloy. ProB is a model checking tool for the B programming language, helping the developer by detecting errors in B specifications [13]. Similarly, Alloy is a language for describing structural properties [11]. Both languages are thus suited for writing complex access rules policies, free of conflicts. The work of Kolovski et. al., provide a formalization of XACML, using description logic which is the basis for the Web Ontology Language (OWL) [12]. Now, XACML is a widely used and standardized access-control policy language.
In the related work described above, the use of logic programming is limited to either the specification, the design and/or verification of access control policies. The logic programs are merely used as a tool or in the backend of a bigger non-logic-based system. On the contrary, the proposed solution can be used inside logic programs to enforce access control. Provided translation, however, rules written in B, Alloy or XACML can be used by ACoP.
Sartoli et al. use Answer Set Programming (ASP) to implement adaptive access control policies, allowing access control on incomplete policies and imperfect data [16]. This approach is interesting as it can handle exceptional cases and supports dynamic environments where former believes may conflict with new observations. While the policies are specified and handled in ASP, the focus is also in providing support as backend solution towards external systems.
Bruckner et al. present a policy system that allows to compile access control policies in the application logic [4]. An automatically created domain specific language is therefore cross-compiled into the host language. Although the system puts no restrictions on the host language, it is unclear if it transfers to logic programming languages as well.
To the best of our knowledge, ACoP is the first to provide a solution to apply access control to logic programs. Hence, the focus of this paper is on how access control can be enforced on the reasoning of logic programs. It not only allows to control access to data or entities, but also to control access to knowledge (e.g., rules).

Support for logic programming languages. Several logic programming languages exist. The proposed solution is validated by a Prolog implementation, and can be integrated into existing Prolog programs. Nevertheless, the approach may be extended to other logic programming languages as well. Examples are Datalog [14], a subset of Prolog, or the more recent Logica [9], a modern logic programming language for data manipulation. However, while in Prolog the meta-interpreter and policies can be fully written in the language itself, writing a meta-interpreter for Datalog and Logica may require more effort and the possibilities to define access control rules will be more restricted.

3 General Approach

Access control is the act of ensuring that a user only has access to what she is entitled to. It is usually defined in three levels, using an access control policy, a security model and a security mechanism [15]. The *policy* expresses the rules according to which control must be regulated. The *security model* provides a formal model of the policy and its working, and the *security mechanism* defines the low level functionality that implements the controls as formally stated in the model. Logic programs naturally support logic-based formulations of access control policies, providing clean foundations and a high expressiveness. In fact, it merges both access control policy and security model, into a single formal specification of the policy. In the remainder, we make no distinction between both, and will use the access control policy to denote both.

In this work, access control policies are defined at the predicate level, by specifying whether access to the predicate is allowed or denied. To ensure completeness (i.e., in case no authorization is specified), a default policy is used. Whether an open (i.e., default access) or a closed policy (i.e., default access denied) is used, is configured at design time by the policy administrator.

Figure 1 shows the structure of a target logic program, protected by the ACoP system. Queries sent to the ACoP system are resolved by the access control module implementing the security mechanism. The burden of adding access control to a logic program is very limited. Introducing access control to a target program requires no changes to the program itself. It only requires the definition of the access control policies and the configuration of the the access control module.

In the following sections, the policies and the access control module will be defined.

3.1 Access Control Strategy and Policies

The applied access control strategy in ACoP is the following: *deny access as soon as possible*. Therefore, access is already verified before the predicate is being resolved, i.e., preliminary access control. If, based on the defined access control policies, it determines that access is denied, resolution will stop. If it cannot yet determine whether access is denied, an attempt will be made to resolve the predicate. Once the predicate is resolved, access is verified again with the now resolved predicate.

Inaccessible data appears as nonexistent. Thus, the user only has a limited view of the entire knowledge base. Queries to inaccessible data do not return any answers, while queries for accessible data should produce the same results as when no access control is used. Note that this may result in a change of semantics: When no results are retrieved, it may either indicate that no answers to the query exist, or that the user has insufficient rights to access the information.

In general, access control policies use a combination of an *Object*, being the resource to which access is requested, and a *Condition*, defining the constraints that need to hold before access is granted. Based on the type of conditions

that can be specified, different access control models exist (e.g., attribute-based, role-based, rule-based, discretionary or mandatory access control). For instance, conditions may relate to the subject, the current context, the allowed operations, etc.

In ACoP, objects are represented by predicates, with no restrictions on how conditions are defined. In other words, any access control model can be supported. The object of an access policy is either allowed or denied, depending on customised conditions. Basically, allow and deny access policies, are defined as follows:

```
allow(Pred(...)) :- <conditions>.
deny(Pred(...)) :- <conditions>.
```

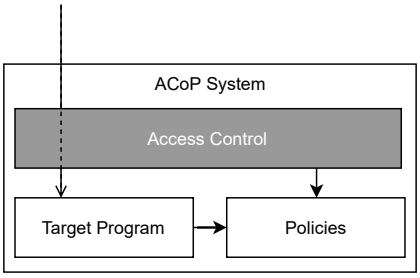

Fig. 1. Structure of the ACoP access control system

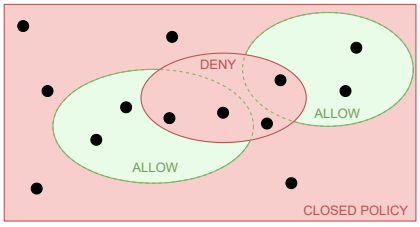

Fig. 2. Venndiagram depicting allowed or denied predicates (•) for a closed policy

Positive and negative policies. The permission of an access policy is either positive (`allow`) or negative (`deny`), granting and refusing access to the predicate respectively. `Pred` is the target predicate (defined or used in the Program) to which the permission applies, i.e. the access rule's object. The predicate may contain a number of atoms as arguments to constrain the applicability of the permissions, others may be left open (i.e., remain variable).

Optional `conditions` define under what circumstances the permission applies. These conditions may contain custom logic, or refer to predicates defined by the target program. Sometimes, permission is not based on the validity of a predicate in a target program, but on whether access to that predicate is granted. Therefore, the predicate `access/1` is introduced. This predicate allows to verify if access is granted to a predicate in the target program.

Multiple permission rules may be defined on the same predicate, both allow or deny, and with different conditions or arguments. The access control module will correctly resolve the potentially conflicting policies, based on the configured access control strategy.

By supporting both allow and deny policies, ACoP allows for more fine-grained rules, in contrast to whether only one type of permission can be specified. In Listing 1.1, the specification of some example access control policies is given

```
% policy 1
allow(machine(M)) :- user(U), line_manager(U,P), location(M,P).
% policy 2
allow(start_machine(M)) :- access(machine(X)).
% policy 3
deny(start_machine(M)) :- night_time.
```

Listing 1.1. Example access control policies in a manufacturing environment.

for a manufacturing environment. The predicate `user/1` requests the identifier of the entity issuing the request to access the object of the policy. The policies are the following. In general, a machine `M` is accessible to the manager of the production line in which the machine is located (policy 1). Starting a machine `M` is permitted when access to the machine itself is allowed (policy 2). However, starting a machine during night time is prohibited (policy 3). Hence, policy 3 further restricts policy 2.

Completeness. For *completeness*, it is required to resolve authorization when no permissions are defined. Therefore, ACoP can be configured for either an *Open* or a *Closed* policy. In an open policy strategy, access is granted by default, while in a closed strategy, access is denied. In traditional access control systems, closed policies are custom as a fail-safe alternative when no permission is defined. However, in logic programs, it could make sense to use an open policy. For instance, in a reactive system controlling a robot, all reasoning is by default allowed, except for the impure predicates that are used to control the robot.

Conflict resolution. To ensure consistency, proper conflict resolution is required. The meaning and resolution of permissions depends on the strategy in use. In a closed policy, one should define allow policies to provide access to predicates. In other words, accessible predicates must be 'allow listed'. To support a more fine grained allow listing, deny policies may overrule accessible predicates. As shown in Figure 2, a deny policy may partially overrule one or more allow policies at once. The opposite reasoning applies for an open policy. Deny policies restrict access to predicates (deny listing). Analogous to the closed case, allow policies may overrule the deny policies, and make access less stringent. This also defines how conflicts are resolved. When access is granted by default, this is also what takes precedence in case of a conflict. Contrarily, when access is denied, denial takes precedence.

Controlled Reasoning. In traditional systems, access control only applies to resources. In contrast, in logic programs, access control can be extended towards its logic rules. When no explicit permissions were found for a certain predicate, ACoP can be configured to infer permissions based on logic rules defining the predicate. In the following, this will be denoted as *body resolution*. This is achieved by scanning the knowledge base for clauses that define the predicate `P`. Permissions for `P` are inferred from the predicates defining `P` (i.e., the body). Permissions for the defining predicates may also be derived from their definition, making access control a recursive process.

Compound statements. Access on a compound statement depends on the accessibility of the predicates in the statement. Therefore, a conjunction of predi-

cates is allowed when each predicate is accessible, while in a disjunction at least one of the predicates must be accessible. This also reflects what would happen in the 'absence' of certain knowledge.

Impure logic and the IoT Environment. A straightforward approach to integrate access control into logic programming would be to check for each resolved predicate used during inference, whether it is allowed to be accessed, and only proceed if it is. Otherwise, it fails and proceeds by backtracking. This approach would work in pure logic, but fails as soon as *impure predicates* are involved. The problem with impure predicates is that during resolution of the predicate, side effects can take place which cannot be undone during backtracking.

Since access control is particularly relevant for applications in the IoT space, e.g., sending instructions to a robot, controlling an actuator in a house, etc., it is important to cover this case. Applying access control should be transparent and handle a query as if the knowledge were absent. Access to an impure predicate must therefore be checked before side effects can take place. Hence the preliminary access control and the default strategy to deny access as soon as possible.

3.2 Terminology and Working Example

Before elaborating the process step by step,the terms *subsumption*, *unification* and *resolution* that are often used in the context of logic programming are explained in more detail. A working example in the field of smart manufacturing, used in Section 3.3, is presented.

subsumption. A predicate A subsumes a predicate B if the predicate A can be made equivalent to B by only instantiating variables in A. For example `location(M,P)` subsumes `location(m1, P)` as the former can be made equivalent to the latter by only instantiating variable `M` to `m1`. The predicate `location(M,p1)` does not subsume `location(m1,P)` because, in order to make the predicates equivalent, also variables in the second predicate must be instantiated.

unification. A predicate A is unifiable with a predicate B if A can be made equivalent to B by instantiating variables in A and/or B. Similarly a predicate A is unified to predicate B if all variables are instantiated to make A equivalent to B. The predicate `location(M,p1)` is unifiable with `location(m1,P)` by instantiating the variable `M` to `m1` and variable `P` to `p1`. After unification both predicates are equal (i.e. `location(m1,p1)`).

resolution. During resolution of a predicate, a logic program recursively searches for terms in the knowledge base that unify with the predicate. After resolution the predicate is resolved. If no unifications can be found, resolution fails. For impure predicates, this is also the moment that side effects occur.

working example. Listing 1.2 presents the working example. It consists of machines located in production lines controlled by a line manager. The impure predicates in this example are `start_machine/1`, which sends a request to start a machine, and `request_state/2` to request the current state (on/off) of a machine.

```
user(alice).
production_line(l1).      machine(m1).          location(m1, l1).
production_line(l2).      machine(m2).          location(m2, l1).
line_manager(bob, l2).    machine(m3).          location(m3, l2).
line_manager(alice, l1).

start_production_line(P) :- production_line(P), location(M,P),
  ↪  start_machine(M).
machine_state(M,S) :- machine(M), request_state(M, S).
% ACCESS CONTROL POLICY 1
allow(location(_,_)).
% ACCESS CONTROL POLICY 2
allow(machine(M)) :- user(U), line_manager(U,P), location(M,P).
% ACCESS CONTROL POLICY 3
allow(start_machine(M)) :- access(machine(M)).
% ACCESS CONTROL POLICY 4
allow(machine_state(M,_)) :- user(U), line_manager(U,L), location(M,L).
```

Listing 1.2. Working Example.

3.3 ACoP Mechanism

To integrate the above access control strategy, ACoP defines a security mechanism, able to enforce permissions on predicates. The rules are applied dynamically during logic inference of a query. It intervenes the normal execution by verifying access during resolution. Figure 3 visualizes the logic used to resolve a

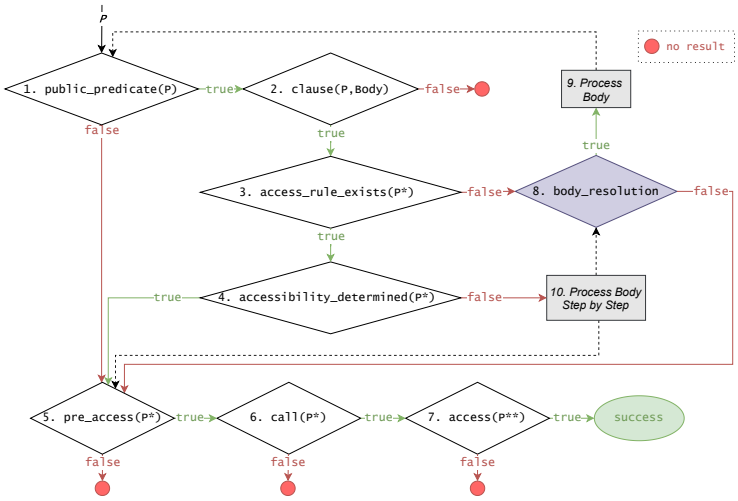

Fig. 3. Simplified flowchart of the ACoP Mechanism

single predicate based on its access control policies. The same scheme is used for both the open and closed policy strategy, and with or without body resolution. Each step for resolving a predicate `P` while enforcing access control is explained below. In order to elucidate the procedure, some steps are demonstrated using the example in Listing 2.

1. `public_predicate(P)`. The first step filters impure and private built-in predicates from public facts and rules present in the knowledge base. For impure and private predicates, examining clauses (step 2) will be unsuccessful, furthermore, body resolution is not meaningful. Thus, for these predicates, control is passed to step 5. For public facts or user defined rules, control is passed to step 2.

   Example 1. The predicate `start_machine(m1)` is an impure predicate and will be forwarded directly to step 5.
   The predicates `machine(M)`, `machine_state(M,S)` and `start_production_line(P)` can be mapped to public facts or rules in the knowledge base, and are forwarded to step 2.

2. `clause(P, Body)`. The second step searches for clauses (i.e. facts and rules) in the knowledge base matching the predicate `P`. When a clause is found, `P` is unified with the head of that clause (denoted as `P'`). In case of rules, `Body` is unified with the body of the rule. For facts, `Body` is unified with the atom `true`. Hence, `Body` is not yet resolved and potential side effects do not take place. Alternative clauses are handled on backtracking. If no clause can be found, resolution stops.

   Example 2. Searching for clauses that match `machine(M)`, results in `P` being unified to `machine(m1)` (i.e. `P'`), and after backtracking `machine(m2)` and `machine(m3)`. In all cases `Body` is unified to `true`. The predicate `machine_state(M,S)` is unified to `machine_state(M,S)`, hence the predicate does not change. `Body`, however, is unified with the compound term `(machine(M), request_state(M,S))`.

3. `access_rule_exists(P')`. The third step checks whether or not an access rule exists for which the target predicate is unifiable with predicate `P'`. This depends on the predicate name and the arguments of the predicate under evaluation (i.e. `P'`). A matching access control policy exists if the predicate under evaluation can be unified to the predicate in the policy. If at least one match for predicate `P'` can be found, further action is taken in step 4. If no access rules match the predicate, access cannot be decided by the provided access rules and control is passed to step 8. The predicate will then either be resolved based on body resolution or determined by the default policy.

   Example 3. Policy 2 of the working example matches predicate `machine(m1)`, as the policy's target predicate, `machine(M)` can be made equivalent to the predicate `machine(m1)`.
   For `start_production_line(l1)`, however, no matching access rule can be found. In that case, the predicate, together with its matching `Body`, is forwarded to step 8.

4. `accessibility_determined(P')`. This step checks whether or not access to `P'` can already be determined based on the defined access rules. It checks whether the target predicate of each matching access rule (determined in step 3) subsumes the predicate `P'`. In addition, ACoP verifies that no variables are present

in both the head and body of the access rule. Those variable terms must be instantiated before access can be properly determined. In case accessibility is determined, step 5 grants or denies access to the predicate. If accessibility is not yet determined (i.e., an access rule matches but does not yet subsume `P'`), the body of the rule will be examined until access is determined in step 10.

Example 4. Accessibility for `machine(m1)` can be determined, as the target predicate of policy 2 (i.e. `machine(M)`) can subsume `machine(M)` and all arguments are sufficiently instantiated to determine access. Access to query all machine states (i.e. `machine_state(M, S)`) cannot be determined yet. Although the target predicate of policy 4 (i.e. `machine_state(M,_)`) subsumes `machine_state(M, S)`, the variable `M` must be instantiated before access can be determined.

5. `pre_access(P')`. This step preliminary verifies access to the predicate before resolution. This check only fails if it is sure that access to the predicate is denied. Otherwise, the predicate is handed to step 6. Note that the default access control strategy, either open or closed, is taken into account if no access rules match. In case of pure logic, preliminary access control is only useful to stop prematurely, omitting this step would not affect the obtained results. In impure logic, however, this step prevents impure predicates to be executed if not allowed, as the occurred side effects can not be rolled back on backtracking.
6. `resolve(P')`. In this step, the predicate `P'` is resolved. In case of a rule or fact, this means that `Body` defined in step 2 is resolved. In case of an impure or private predicate, the predicate itself is resolved and possible side effects take place. As in execution without access control, when resolving the predicate fails, resolution fails. Otherwise, `P'` is resolved (denoted as `P''`) and is further handled in step 7.
7. `access(P'')`. Similar to step 5, permission to access `P''` is verified. This second iteration is required since additional arguments in the predicate might be instantiated, and certain policies may become applicable. Note that the default access control strategy is also taken into account here. If access is still allowed, the resolution of the predicate ends successfully, else it fails.
8. `body_resolution`. When no explicit permissions are defined for the predicate, this step consults the configuration and checks whether `body_resolution` is active. If that's the case, it proceeds to step 9. Otherwise the predicate is passed to step 5, that takes the default strategy into account to decide upon access to the predicate.
9. *Process Body.* In this block, access to the predicate `P'` is decided based on the body of the clause found in step 2. It does this with body resolution, by repeating the entire process for each predicate in `Body`, taking into account access control in compound statements as described in Section 3.1 .

Example 5. The predicate `start_production_line(l1)` with body `(production_line(l1), location(M,l1), start_machine(M))` ends up in this step. The entire process is thus repeated for the compound term `(production_line(l1), location(M,l1), start_machine(M))`. As a result all machines in production line will unlock, with the condition that the user has the authority to do so.

10. *Process Body Step by Step.* When a matching access rule exists for the predicate (step 3), but access cannot yet be determined (step 4), the terms in `Body` are resolved step by step. Processing the body step by step causes variables to be instantiated leading to one of the events below, causing the processing to stop.
    (a) *Access to the predicate `P'` becomes determined.* Enough terms in `Body` are resolved such that variables in `P'` become instantiated and allow to determine the accessibility to the predicate based on the defined access rules. In other words, there exists an access rule's predicate that subsumes predicate `P'`. Accessibility will then be decided in step 5.
    (b) *The access rules no longer apply.* It is possible that by resolving terms in `Body`, the arguments are instantiated such that there are no more access rules for which the target predicate matches with predicate `P'`. Access to predicate `P'` can then still be decided using body resolution (step 8 and 9).
    (c) *The body is entirely executed.* If after resolving the entire body, the matching access rules are still not subsumable, and thus will never be. Access must be decided using body resolution if applicable (step 8 and 9).

    Special care must be taken when `Body` contains impure predicates, because side effects cannot be reversed. Therefore, an additional access control check is performed on the original predicate before resolving impure predicates during the step by step processing of the body. Impure predicates are thus only executed if access is granted. It is important to keep observing the original predicate to check when one of the above events occurs, as well as to keep track of the terms that have already been resolved. As a term in the body of a rule might be defined by a rule itself, processing the body step by step is a recursive process. After processing the body, already resolved terms must be taken into account such that already executed impure predicates, and corresponding side effects, are not executed twice.

Example 6. The predicate `machine_state(M,S)` with body `(machine(M), request_state(M, S))` is handled here. To begin, the first term of the compound body (i.e. `machine(M)`) is resolved, resulting in the variable `M` being instantiated to `m1`. Consequently the original predicate is instantiated to `machine_state(m1,S)`. Now it is necessary to check again whether access to the predicate can be determined (step 4). As the predicate is sufficiently instantiated, and policy 4 subsumes the predicate, access can be decided. The predicate is forwarded to step 5, where the access is determined. As the current user is line manager of the production line where machine `m1` is part of, access is granted and the predicate can be resolved (step 6). The state of the machine is requested and returned to the user.

To generalise the case of access control from a single predicate to compound statements (used in both user queries and logic rules), ACoP applies the rules discussed for compound statements in Section 3.1. However, in a conjunction, the resolution of predicates that come later may instantiate variables. As a result, certain access rules may become applicable later. Therefore, an additional access control check is performed on the predicates earlier in the chain to ensure that access is still granted.

4 A Prolog Implementation of ACoP

To validate the security mechanism discussed in Section 3, an implementation is available for SWI-Prolog. Access control is enforced by a Prolog meta-interpreter that can be plugged in and configured in any Prolog program. The implementation can be found at `https://github.com/ku-leuven-msec/ACoP`. In this section, we will take a closer look at some of the implementation details.

4.1 Implementing the Access Control Logic

The meta-interpreter takes advantage of query expansion to replace normal query resolution with inference that includes the ACoP's access control logic. This allows an almost plug-and-play use of access control in an existing program. Access control is transparent to the user and queries send to the reasoner automatically resolve with access control in place.

In the previous section, the steps required to implement the security mechanism have been discussed. The implementation of the most important constructs in Prolog are now discussed in more detail.

public_predicate. As described in Section 3.3, the first step is to separate public predicates from private and impure predicates. This is done using the *predicate_property/2* predicate which provides the properties of a given predicate. In the proposed implementation, predicates with the *built_in* or *foreign* property are defined as private, resp. impure predicates. While the former specifies built-in predicates for which no body can be retrieved, the latter defines predicates that have its implementation defined in the C-language. The execution of such predicates often result in side effects (i.e. impure predicates).

access. The implementation to determine access builds upon the basic predicate `subsumes_chk/2` which checks if a predicate can be subsumed by another given predicate.

To determine access to a predicate, two additional rules are defined. The `match_allow/1` and `match_deny/1` predicates, as shown in Listing 1.3, verify whether there is a definition for a positive, resp. negative permission that matches the predicate `P`. A policy matches a predicate only if the predicate in the policy (`Pol`) is more generic or equivalent to `P` (i.e., using `subsumes_chk/2`). The `access` predicate for both the open and closed policy strategy is presented.

```
% Matching allow, resp. deny policies.
match_allow(P) :- copy_term(P,Pol), allow(Pol), subsumes_chk(Pol,P).
match_deny(P) :- copy_term(P,Pol), deny(Pol), subsumes_chk(Pol,P).
% Open policy
access(P) :- (match_allow(P); \+match_deny(P)), !.
% Closed policy
access(P) :- (match_allow(P), \+match_deny(P)), !.
```

Listing 1.3. Checking if access to a predicate is allowed based on current knowledge.

In the *open policy* case, it looks for a matching allow or the absence of a matching deny rule. Hence, access is allowed if there is *a matching allow or no*

matching deny. It fails only if there is no matching allow and a matching deny policy.

In the *closed policy* case, reasoning is slightly different, and requires *a matching allow policy and no deny* to be successful. If there is no allow rule or there is a deny rule that applies to the predicate `P`, access is denied. When a predicate matches multiple policies, backtracking is not desired, therefore, the cut-operator prevents alternative resolutions for granting access.

Process Body Step by Step. When processing the body of a rule step by step, as described in Section 3 the current state is tracked. The state keeps track of the terms that have already been resolved and the terms that have not yet been resolved, in order to prevent duplicate resolution of impure predicates. Therefore the predicate `state` is introduced and defined as follows:

```
state(Predicate, Resolved, ToResolve).
```

`Predicate` is the head of the rule and also the predicate under evaluation, `Resolved` is a list of the resolved terms and `ToResolve` is a list of the terms that are not yet resolved. As a term in the body of a rule might be defined by a rule itself, processing the body step by step is a recursive process and the `Resolved`-list can also contain states of terms. The original predicate for which access must be defined is being monitored after every step, either until access can be decided, the existing access rules are no longer applicable, or the body is completely resolved. The final state is handled depending on how the processing ended. If access is determined and allowed, all terms in `ToResolve` are being resolved. If the existing rules are no longer relevant or the body is entirely processed, body resolution can still be used to decide access. In case body resolution is applicable, it is first checked whether access to the previously resolved terms is allowed. If access is allowed, all terms in `ToResolve` are processed taking into account the access policy.

4.2 Evaluation

Figure 4 shows the execution time for the query `?- machine(M)`, in function of the machines per production line. The smart manufactory as described in Listing 1.1 and consists of 3 managers each controlling 5 production lines. Adding access control clearly has implications to the performance of the logic program. The current implementation is built to support different scenarios, but does not yet include major optimizations. Nevertheless, it is clear that the way access control policies are defined only has a linear impact on the performance of the program.

Figure 5 shows the amount of inferences for different queries in the previously presented manufactory setting with ten machines per production line. Several conclusions can be made based on the four performed queries.

– For queries on predicates for which access rules exist (i.e. query 1, 2 and 3), there is no difference in a setup with or without body resolution. Since matching access rules can be found, body resolution must not be performed. Thus, the execution time is independent from whether body resolution is enabled or not.

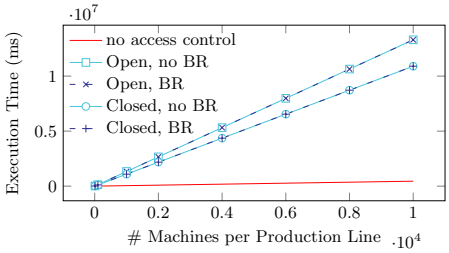

Fig. 4. Execution time for a variable number of predicates

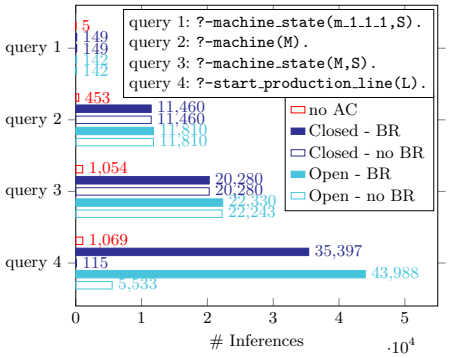

Fig. 5. Amount of inferences for different queries in different setups

- The more specific a query is, the smaller the overhead caused by access control. This is because the amount of variables to instantiate is lower (i.e. query 1 versus query 3).
- The amount of inferences for an open policy are different than for a closed policy. This is both a result of how access control is handled and how the policies are defined. For the open policy, ACoP can already stop resolution if at least one matching allow rule can be found. For the closed policy, however, not only an allow rule must be found but all deny rules must be verified to be sure access is allowed. Access control could therefore be determined more quickly in the case of an open policy, which is the case for query 1. For this example, however, the open policy rules are very basic and equal to the closed policy rules complemented with a deny rule without conditions on `machine/1`, `start_machine/1` and `machine_state/2`. This results in an increased overhead for the open policy and becomes more apparent when more steps in the ACoP process have to be taken (i.e. queries 2, 3 and 4).
- The amount of inferences for query 4 depends on the body resolution setting. There is no matching access control rule for this query. In case body resolution is disabled, ACoP can quickly decide whether or not to resolve the predicate, with little impact on the query. When body resolution is enabled, access must be controlled for each predicate in the body of the rule, which quickly increases the number of steps. Note that for this query the closed policy without body resolution is the only case for which access control is denied, resulting in a lower number of inferences than when access control is disabled.

5 Discussion

While access control to resources is well-studied, applying this to predicates and rules is not straightforward. Below, we discuss a number of aspects that need careful treatment.

Filtering on 'outputs' - When an access control policy on an impure predicate filters on 'output' arguments (i.e., arguments that only get instantiated after resolution), it implies that the predicate is resolved (i.e. side effect take place)

```
age(X,A) :- info(X,birthdate,date(Y)), calculate_age(Y,A).
allow(age(_,A)) :- A>18.
?- age(X,Y).
```

Listing 1.4. Example of an insufficient instantiation error.

before it is denied access. A warning during consultation time could inform the developer of such cases, to make adjustments to the policies, if necessary. Note that the reasoner must know the output argument, which can be accomplished by annotating the output arguments during development time.

access(...) - The `access` predicate may be used in the body of a rule to verify whether access to another predicate is allowed. This may possibly lead to infinite loops. Hence, special care must be taken when this predicate is used. Especially when body resolution is active.

Insufficient instantiation - Often, access control policies filter on the values of arguments. In complex cases, one of the arguments of the predicate under control may be used to compare with some value. An example is shown in Listing 1.4. Although this seems intuitively correct, the query results in an error. Since access control is also verified before resolving the predicate, the arguments of the predicate may still be variable (both for pure and impure predicates). Using variables, while non-variables are expected may result in unexpected behavior. Since it cannot be derived from the predicates whether arguments are allowed to be variable, it is not possible to verify this automatically.

To handle this, either the arguments used in the filter must be instantiated properly, or the developer needs to take additional measures when defining the rules. For instance, `nonvar/1` can be used to check whether a term is already instantiated, before performing arithmetic operations. Another solution to solve such problems is by giving the possibility to ignore the argument during preliminary access control. For instance, by annotating such arguments.

Support and conflict resolution - As discussed in Section 3.1, conflict resolution is determined by the default policy. In a closed policy, deny rules take precedence, while in an open policy, allow rules take precedence, even in the presence of body resolution. This makes it possible to write access rules that will not be considered, but give the developer an unjustified feeling of control. Adding support to track and warn users of aforementioned cases could prevent the misleading feeling of security.

Data Privacy - Although access control may help in preventing access to specific information, it does not prevent that rules may still leak information. Finding solutions to prevent such leakage is left for future work.

Supporting Access Control Strategies Since ACoP does not constraint the conditions in policies, it naturally supports various access control strategies. While for Identity Based Access Control (IBAC), a policy may verify the identity of the current user or her membership in an access control list, Role Based Access Control (RBAC) may go one step further and check the user for required roles. Similarly, supporting Relationship Based Access Control (ReBAC), often used in the context of social networking systems, simply requires a check whether the

current user has a certain relation to the owner of some asset. Some examples may be found in A.

6 Conclusions

This paper presented ACoP, an access control mechanism that enforces access control in existing logic programs. Access control is fine grained on the level of predicates, supporting multiple established access control strategies. The solution takes into account the use of impure predicates and applies a *deny as soon as possible* strategy to prevent prohibited side effects from taking place. The presented Prolog meta-interpreter shows that the integration does not entail excessive overhead.

References

1. Abadi, M.: Logic in access control. In: 18th Annual IEEE Symposium of Logic in Computer Science, 2003. Proceedings. pp. 228–233. IEEE (2003)
2. Bohé, I., Willocx, M., Lapon, J., Naessens, V.: Towards low-effort development of advanced iot applications. In: Proceedings of the 8th International Workshop on Middleware and Applications for the Internet of Things. pp. 1–7 (2021)
3. Böhme, R., et al.: A fundamental approach to cyber risk analysis. Variance **12**(2), 161–185 (2019)
4. Bruckner., F., et al.: A framework for creating policy-agnostic programming languages. In: Proceedings of the 9th International Conference on Data Science, Technology and Applications - DATA,. pp. 31–42. INSTICC, SciTePress (2020)
5. Chin, B., et al.: Yedalog: Exploring knowledge at scale. In: 1st Summit on Advances in Programming Languages (SNAPL 2015). pp. 63–78. Dagstuhl, Germany (2015)
6. Edwards, B., et al.: Hype and heavy tails: A closer look at data breaches. Journal of Cybersecurity **2**(1), 3–14 (2016)
7. Ferraiolo, D., Kuhn, D.: Natl institute of standards and tech., dept. of commerce, maryland, role-based access control. In: Proceedings of 15th Natl Computer Security Conference (1992)
8. Gates, C.: Access control requirements for web 2.0 security and privacy. IEEE Web **2**(0), 12–15 (2007)
9. Google Open Source: Logica. `https://opensource.google/projects/logica` (2021), (accessed: 05.05.2021)
10. Huynh, N., Frappier, M., Pooda, H., Mammar, A., Laleau, R.: Sgac: A patient-centered access control method. In: 2016 IEEE Tenth International Conference on Research Challenges in Information Science (RCIS). pp. 1–12 (2016). https://doi.org/10.1109/RCIS.2016.7549286
11. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (Apr 2002)
12. Kolovski, V., et al.: Analyzing web access control policies. In: Proceedings of the 16th international conference on World Wide Web. pp. 677–686 (2007)
13. Leuschel, M., et al.: Prob: A model checker for b. In: FME 2003: Formal Methods. pp. 855–874. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
14. Maier, D., et al.: Computing with Logic: Logic Programming with Prolog. Benjamin-Cummings Publishing Co., Inc., USA (1988)

15. Samarati, P., et al.: Access control: Policies, models, and mechanisms. In: Foundations of Security Analysis and Design. pp. 137–196. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
16. Sartoli, S., et al.: Modeling adaptive access control policies using answer set programming. Journal of Information Security and Applications **44**, 49–63 (2019)
17. Sterling, L., et al.: The art of Prolog: advanced programming techniques. MIT press (1994)
18. De Capitani di Vimercati, S.: Access Control Policies, Models, and Mechanisms, pp. 13–14. Springer US, Boston, MA (2011)

7 Appendices

A Example Support for Access Control Strategies

ACoP can easily be used to enable various access control strategies. Several examples of how possible strategies can be implemented can be found in this section.

Identity Based Access Control One of the most basic access control strategies is Identity Based Access Control (IBAC). Access to a resource is determined based on the identity of the individual trying to access the resource. An IBAC strategy for accessing files can be achieved using ACoP with the using the rule,

```
allow(file(<filename>) :- current_user(<user_identifier>).
```

in a closed policy. For example can access to *file1.txt* be granted for both *Alice* and *Bob* in the following way:

```
allow(file(file1.txt) :- current_user(alice).
allow(file(file1.txt) :- current_user(bob).
```

In IBAC, an access control list (ACL) is often used to bundle all identifiers together. To support use of an ACL, the ACoP access rule can be defined as follows, assuming that `acl(L)` unifies `L` with a list of the identifiers that may access he resource:

```
allow(file(file1.txt) :- current_user(U), acl(L), member(U,L).
```

Role Based Access Control Role based access control (RBAC) was first introduced by Ferraiolo et. al. in 1992 [7]. It is since a widely used strategy in large companies and as the name states based on roles assigned to users of the system. An example for the role based access control strategy, is a blogpost website, where dependent on the role, a user can take several actions. The limited set of possible roles and actions that can be taken on the blogpost website can be found in Table 1.
To enable RBAC using ACoP, the users of the system must be defined together with their role. The closed policy used for the blogpost website will then be:

Table 1. Blogpost Website: Roles and Actions

	Visitor	Subscriber	Admin
Add/Remove User			✓
Publish Posts			✓
Comment on Posts		✓	✓
Read Posts	✓	✓	✓

```
user_role(alice, admin).
user_role(bob, subscriber).
user_role(_, visitor).
```

Defining the actions that can be taken by each role can easily be done as follows:

```
allow(add_user(_)) :- current_user(U), user_role(U,admin).
allow(publish_post(_)) :- current_user(U),user_role(U,admin).
allow(post_comment(_,_)) :- current_user(U),
  ↪  (user_role(U,admin);user_role(U,subscriber)).
allow(read(P)).
```

Relationship Based Access Control Relationship based access control (ReBAC) was first introduced by Gates in 2007 [8]. Access control policies to resources are defined based on relationships between users, and are mainly used in the context of social networking systems. Figure 6 gives an example of a social networking system with friend relations and personal information and photo resources.

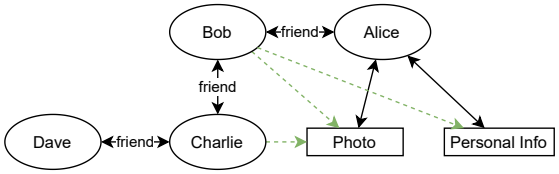

Fig. 6. Social network system with friend relations

The access control policy is the following. A user has access to a users personal information, if they have a friend relation. A user has access to a users photos, if the person is a friend of the owner (i.e. has a friend relationship), or if they have a friend relation with a friend of the owner. To enable the ReBAC policy, using the ACoP system, users must be defined together with the resources they own as well as the relationships between the users. The policy can then be described as follows, using a closed policy:

```
allow(personal_info(I)) :-  owner(O,I), current_user(U), friend(U,O,
  ↪  friend).
allow(photo(P)) :- owner(O,P), current_user(U), (relation(U,O,
  ↪  friend); (relation(U,F,friend),relation(F,O, friend))).
```

Note that the `relation/3` predicate could also be simplified and defined using a `friend/2` predicate.