Representing Flexible Role-Based Access Control Policies Using Objects and Defeasible Reasoning

Reza Basseda[1(✉)], Tiantian Gao[1], Michael Kifer[1], Steven Greenspan[2], and Charley Chell[2]

[1] Computer Science Department, Stony Brook University, Stony Brook, NY 11794, USA
{rbasseda,tiagao,kifer}@cs.stonybrook.edu
[2] CA, Inc., 520 Madison Avenue, New York, NY 10022, USA
{steven.greenspan,charley.chell}@ca.com

Abstract. Access control systems often use rule based frameworks to express access policies. These frameworks not only simplify the representation of policies, but also provide reasoning capabilities that can be used to verify the policies. In this work, we propose to use defeasible reasoning to simplify the specification of role-based access control policies and make them modular and more robust. We use the Flora-2 rule-based reasoner for representing a role-based access control policy. Our early experiments show that the wide range of features provided by Flora-2 greatly simplifies the task of building the requisite ontologies and the reasoning components for such access control systems.

Keywords: Access control policy · Object oriented logic programming · Ontology

1 Introduction

Administering and maintaining access control systems is a challenging task, especially when the environments are complex and the authorization requirements are subject to frequent change. Policy languages play an important role in designing and implementing flexible access control systems. There are a number of role-based policy specification languages that can express access control policies, including XACML [14], X-RBAC [8], Rei [9], Common Policy [18], and Ponder [4]. These languages simplify management of access control by factoring the authorization policy out of the hard-coded resource guard. For example, XACML defines a general XML role-based policy language. X-RBAC is another XML role-based access control language for specifying RBAC policies [8]. Common Policy provides a framework for authorization policies controlling access to application-specific data. Although they have been applied to a broad domain of enterprise environments, it is difficult to use them at the semantic level. To mitigate this problem, some approaches inject RDF [2] and OWL [6] into the mix.

DOI: 10.1007/978-3-319-21542-6_24

Rei [9] is an example of this approach, which uses OWL-Lite and RDF. Ponder is a declarative policy language, which is designed based on object-oriented principles. It can be used to specify both security and management policies. Ontologies are also used to develop hybrid distributed access control systems [19]. Another access control policy specification language based on OWL and SWRL has been proposed in [13].

Clearly, representation of domain classes and objects is a key component of an access control system because such representation facilitates the development and changing of policies. Although all of the above mentioned access control policy specification languages are able to represent domain classes and objects, none is a *rule*-based language and none is as expressive as a rule based policy specification language can be.

One of the key challenges in role-based access control is that policies are subject to frequent changes, which calls for hierarchical structure of policy components and roles. However, none of the above mentioned languages can matches the flexibility for defining such hierarchies that is provided by object-oriented rule-based languages based on F-logic [11,12], such as Flora-2 [10,22]. The use of such an expressive knowledge representation and reasoning language lets us both to integrate hierarchies between policy components into policy rules and encapsulate different components of rule based policies in different modules. This gives us the necessary machinery to localize the changes initiated by the clients. The reasoning capabilities that come with object-oriented rule languages like Flora-2 also allows one to make policies more concise, clearer, easier to specify, analyze, and change. Modularity also helps with certain security issues. For instance, in many applications, especially in distributed systems, rules and facts used for access control decision making are ranked and grouped by their trust levels. The reasoning mechanism that understands encapsulation can take into account the different levels of trust when it responds to access requests.

In this paper, we show that by using an elegant defeasible reasoning system we can build a rule based access control policy in terms of separate, encapsulated modules based on the application semantics and security requirements. We use *Logic Programming with Defaults and Argumentation Theories* (LPDA) [20] to define different groups of rules and facts and use this logic to make our access control decision. Together with the higher-order features of Hilog [3] and object-oriented nature of F-Logic [12], great flexibility is provided to the access control policy developers.

The rest of the paper is organized as follows. Section 2 provides a brief overview of logic programming with defaults and argumentation theories. Section 3 illustrates our methodology for building a flexible access control system and its corresponding architecture. Section 4 gives a practical example of defining different components in the rule based access control policy, and Section 5 concludes the paper.

2 Overview of Defeasible Reasoning

Defeasible reasoning is a type of non-monotonic reasoning where conclusions may have priorities and be *defeated* by other conclusion. Such theories are usually conducive to specifying general defaults and conclusions can be easily, modularly, and incrementally altered when new information becomes available. This contrasts with monotonic logic where any previously inferred information remains valid with the addition of new knowledge. For example, given the access control policy stating that *typically, a student is authorized to use a device unless the student has abused the device before,* and the facts that *John is a student and a printer is a device,* we might conclude that John is authorized to use the printer. However, if later it becomes known that *John has abused the printer*, the previous conclusion can be defeated without making any modifications to the policy. Defeasible reasoning is intended to model this kind of scenarios in modular and natural fashion.

General non-monotonic resoning frameworks, such as circumscription, default logic, and autoepistemic logic, can also model the above scenarios, but their languages are not attuned to making changes modular and simple. In this work, we use Logic programming with defaults and argumentation theories (LPDA), a unifying defeasible reasoning framework that uses *defaults* and *exceptions* with prioritized rules, and argumentation theories. LPDA is based on the three-valued well-founded semantics [17]. Here we briefly review LPDA. Defails can be found in [20].

A *literal* has one of the following forms:

- An atomic formula.
- $neg\, A$, where A is an atomic formula.
- $not\, A$, where A is an atom.
- $not\, neg\, A$, where A is an atom.
- $not\, not\, L$ and $neg\, neg\, L$, where L is a literal.

Let A be an atom. A *not-free* literal refers to a literal that can be reduced to A or $neg\, A$. A *not*-literal refers to a literal that can be reduced to $not\ A$ or $not\ neg\, A$. LPDA has two types of rules: *strict* and *defeasible*, where *strict* rules generate non-defeasible conclusions and *defeasible* rules generate defeasible conclusions that can be defeated by some exceptions. A strict rule is of the form:

$$L \leftarrow Body$$

where L is a *not-free* literal and $Body$ is a conjunction of literals. A defeasible rule is of the form:

$$@r\ L \leftarrow Body$$

where r is a *term* that denotes the label of the rule.

Each LPAD program is accompanied by an argumentation theory that specifies when a defeasible rule is defeated. An argumentation theory is a set of definite rules with four special predicates: `\defeated`, `\opposes`, `\overrides`, and `\cancel` where `\defeated` denotes the defeatedness of a defeasible rule, `\opposes` indicates the literals that are incompatible with each other, `\overrides` denotes a binary relation between defeasible rules indicating priority, and `\cancel` cancels a defeasible rule. There can be several different argumentation theories that can be used simultaneously for different modules. Users can select one of the predefined ones and use it as is or modify it, as appropriate. A rule is defeated if it is *refuted*, *rebutted*, or *disqualified*. The meaning of *refuted*, *rebutted*, and *disqualified* depends on the chosen argumentation theory. Generally, a rule is *refuted* if there is another rule that draws an incompatible conclusion with higher priority. A rule is *rebutted* if there is another rule that draws an incompatible conclusion and there is no way to resolve the conflict based on the relative priorities. A rule is disqualified if it is *cancelled*, self-defeated, etc. An example is shown in Figure 1.

```
@{id1} authorized(?Principal,?Dev) :- device(?Dev),
                                      principal(?Principal).

@{id2} \neg authorized(?Principal,?Dev) :- abused(?Principal,?Dev).

\overrides(id2,id1).
\opposes(authorized(?Principal,?Dev), \neg authorized(?Principal,?Dev)).

principal(Mary).
principal(John).
device(printer).
abuse(John,printer).
```

Fig. 1. An example of a simple LPDA program

In the figure, rule `id1` says that if there is a person and a device, then the person is authorized to use the device. Rule `id2` says that if a person has abused the device, then the person is not authorized to use the device. The predicate `\overrides(id2,id1)` indicates that rule `id2` has higher priority than `id1`. The statement `\opposes(authorized (?Persn, ?Dev),` *neg*`authorized(?Persn,?Dev))` says that one can be either authorized or not, but not both. Taking into account the facts `person(Mary)` and `device(priter)`, we can conclude `authorized(Mary,printer)` from rule `id1`. From the facts `person(John)`, `device(printer)`, and `abuse(John,printer)`, rules `id1` and `id2` derive *contradictory* conclusions that both `authorized(John,printer)` and *neg*`authorized(John, printer)` hold. Since rule `id2` has a higher priority than rule `id1`, `authorized(John,printer)` is defeated.

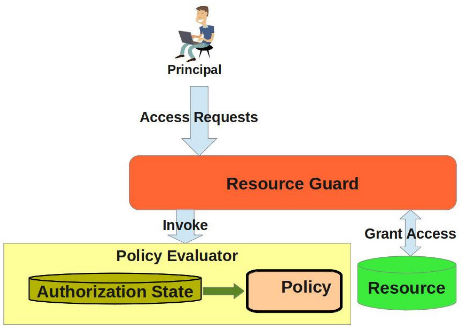


Fig. 2. A typical architecture of an access control system

3 Methodology and Architecture

Although several architectures have been proposed for access control systems [15,16], none of them has gained the status of a standard. To explain different access control policy representation languages, we assume a simple architecture in Figure 2, borrowed from [1]. However, the discussion below applies to more complex architectures as well. To keep our technique as general as possible, we also do not limit our framework to any specific classic access control model, such as Role-Based Access Control [5] or Attribute-Based Access Control [7] Models.

As shown in Figure 2, the authorization *policy* is not hard-coded as a *resource guard* but instead appears as a list of declarative rules. When a principal requests access, the resource guard issues an authorization query to the policy evaluator. Access is granted only if the policy evaluator succeeds in proving that the request complies with the local policy and a set of facts describing the *authorization state*, i.e., with a set of relevant facts, including the knowledge obtained from submitted or fetched credentials. For instance, the history of locations of a principal can be reflected in the authorization state and used by policy evaluator.

This approach greatly increases the maintainability of access control systems, as modifying the declarative policy rules is much simpler than rewriting and recompilation of the code embedded in the resource guard. In fact, resource guards are usually designed to take care of the low-level security considerations while policies are expected to be high-level descriptions of security requirements. Therefore, imperative programming languages (e.g. C or C++) are used to implement resource guard, while higher-level declarative languages are preferred for security policies. There are several reasons why policies should be formally verifiable. For one, the declarative nature of policy languages and the formal framework required for query evaluation make logic programming languages the top candidates for policy specification.

In accordance with this architecture, we assume that the policy evaluator is completely separate from the resource guard. To issue an authorization query to the policy evaluator, the resource guard uses a predicate of form $grantAccess(t_1, \ldots, t_n)$ as a query. Given a set $\mathbb{R}$ of policy rules, the policy evaluator returns *true* or *false* answer, thereby allowing or disallowing the access. Next, we will show how using an object oriented logic programming and defeasible rules can make an access control system much simpler and more flexible.

3.1 Resilience to Changes

Access control policies are not usually considered as a fixed component of an access control system and they are often modified on the request of non-technical policy makers. Therefore, it is very important to make policies as flexible as possible and to minimize the cost of changes. The following features are therefore very desirable:

– Prevention of introduction of bugs through modification via semantic constraints.
– A robust patching mechanism for expansion of policies.

We will now explain how object oriented features in Flora-2 [10,22] and defeasible reasoning via LPDA [20,21] solve these issues.

Classes and Objects: We use a set of classes to represent different resources and roles used by the policy. These classes serve both as semantic integrity constraints and as a policy development guide. The classes are typically identified by IRIs pointing to the actual resources, which is useful for standardization and portability. Figure 3 shows two sample classes in a typical policy represented in Flora-2.

```
Person[|
    firstName => string,
    lastName  => string |].

Employee::Person[|
    employmentYear => integer,
    department     => Department,
    profession     => string,
    rank           => Rank,
    loc(?)         => Location |].
```

Fig. 3. An example of ontology for access control systems in Flora-2

Modification via Patching: To provide a patching mechanism, we use defeasible reasoning to override default rules of a policy with new rules. Consider a policy $\mathbb{P}$ consisting of n rules of the form $@r_i\ L_i \leftarrow Body_i$ where $1 \le i \le n$. Suppose that we need to change $\mathbb{P}$ to $\mathbb{P}'$ such that for some $1 \le j \le n$, a new rule of the form $@r'_j\ L_j \leftarrow Body'_j$ derives L_j, L_j conflicts with L_i, and the new rule has higher priority if condition $Cond$ holds. To obtain $\mathbb{P}'$ out of $\mathbb{P}$, one needs to simply add the following rules to $\mathbb{P}$.

$$\begin{array}{l} @r'_j\ L_j \leftarrow Body'_j. \\ \backslash overrides(r'_j, r_j) \leftarrow Cond. \\ \backslash opposes(L_i, L_j). \end{array} \tag{1}$$

We can also use a similar technique also to disable a rule under certain circumstances. Suppose that for some $1 \leq j \leq n$, we need to disable the rule $@r_j\ L_j \leftarrow Body_j$ from $\mathbb{P}$ when condition $Cond$ is true. To this end, one can simply add

$$\backslash cancel(r_j) \leftarrow Cond. \tag{2}$$

Note that $Body'_j$ may have literals that are defined by other rules in which case those rules would be added as well. The following example illustrates how this patching mechanism works.

Example 1 (Access Control Based on Time and Location). Consider the policy shown in Figure 4. A policy evaluator can use this policy to answer queries of the form $grantAccess(?E, ?R, ?T, ?D)$ where the variables $?E$, $?R$, $?T$, $?D$ range over the members of the classes Employee, Resource, TimeOfAccess, and DateOfAccess, respectively. The first rule defines the predicate $hasmoved(?E, ?D1, ?D2)$ which is true if the location of employee $?E$ is different on day $?D1$ and day $?D2$. The second and third rules define the predicate $moved(+?E, +?D1, +?D2, -?M)$,[1] which binds $?M$ to 1 if the employee $?E$ has moved between days $?D1$ and $?D2$. The predicate $locRisk(+?E, +?D, -?K)$ specifies the security risk if employee $?E$ is known to have moved in each of the four days preceding day $?D$. Finally, the predicate $grantAccess(+?E, +?R, +?T, +?D)$, if true, indicates that the employee $?E$ is allowed to access the resource $?R$ at time $?T$ of day $?D$. This rule just checks if the departments of the employee $?E$ and of resource $?R$ are the same and the risk assessment of the employee is below the threshold.

Suppose that now it is required to use a new parameter called *access time risk*, which computes the risk based on the access hour with respect to $13:00$, *if* the employee is away from the home department. To this end, we construct a patch that enforces the new policy, as shown in Figure 5. The second rule in the figure defines $timeRisk(+?R, +?T, -?TD)$ as the difference between the access time and $13:00$ (this number may indicate the risk of unauthorized accesses). For example an access request at $14:00$ is more reasonable than at at $21:00$ or $03:00$. The third rule says that access is prohibited if the employee is traveling and $timeRisk$ exceeds the threshold. The first fact in Figure 5 states that atom $grantAccess/4$ resulted from rule `locAccess` is defeated by the same atom resulted from rule *timeAccess*. Note that the rule `locAccess` is not completely disabled: it still holds sway if the employee is not traveling.

[1] $+$ indicates that the variable is used as input and must be bound before calling the predicate; $-$ means that the variable is an output and will be bound after calling the predicate produces an answer.

```
hasmoved(?E,?D1,?D2) :-
   ?E:Employee[loc(?D1) -> ?L1],
    ?E[loc(?D2) -> ?L2],
    ?L1 != ?L2.

moved(?E,?D1,?D2,1) :- hasmoved(?E:Employee,?D1,?D2).

moved(?E,?D1,?D2,0) :-
    ?E:Employee,
    \naf hasmoved(?E,?D1,?D2).

locRisk(?E,?D,?K) :-
    ?E:Employee,
    moved(?E,?D,?D1,?M1),
    moved(?E,?D1,?D2,?M2),
    moved(?E,?D2,?D3,?M3),
    moved(?E,?D3,?D4,?M4),
    nextDay(?D4,?D3),
    nextDay(?D3,?D2),
    nextDay(?D2,?D1),
    nextDay(?D1,?D),
    ?K \is ?M1 + ?M2 + ?M3 + ?M4.

@{locAccess}
grantAccess(?E,?R,?,?D) :-
    ?E:Employee[department-> ?DE],
    ?R:Resource[owner-> ?DE],
    locRisk(?E,?D,?K),
    ?K < 3.
```

Fig. 4. An example of a simple policy in Flora-2

```
\overrides(timeAccess,locAccess).

timeRisk(?T,?TD) :- ?TD \is abs(?T - 13).

@{timeAccess}
\neg grantAccess(?E,?R,?T,?D) :-
    ?E:Employee,
    ?R:Resource,
    ?E.department.location != ?E.loc(?D),
    ?E[timeWorked(?D) -> ?T],
    timeRisk(?T,?K),
    ?K > 5.
```

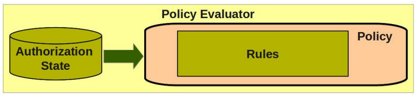
Fig. 5. The first modification of the policy

Now suppose that policy makers suddenly realize that time is different in different time zones, so they decide to calculate access times based on employee's local time rather than resource's local time. This means that the rule `timeAccess` will now be defeated by a new rule, `flexAccess`, if the locations of the resource $?R$ and the employee $?E$ are different. Figure 6 shows the rules of this patch. The third rule defines $timeRisk(+?E, +?T, +?D, -?TD)$, which gets an employee $?E$ and a GMT time id $?T$, computes the actual time in the time zone of the employee, and then assesses the risk according to the employee's local time zone. The `flexAccess` rule for $grantAccess(+?E, +?R, +?T, +?D)$ now says that the employee $?E$ can access resource $?R$ at time $?T$ on day $?D$, if the access happens within the local normal working hours. Other than that, the conditions are the same as for `locAccess`. □

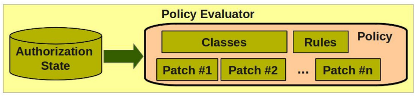
```
\overrides(flexAccess,timeAccess).

timeRisk(?E,?T,?D,?TD) :-
    ?E:Employee[loc(?D) -> ?L],
    ?L[timeZone -> ?TZR],
    ?TD \is abs(?T + ?TZR - 13).

@{flexAccess}
grantAccess(?E,?R,?T,?D) :-
    ?E[department-> ?DE],
    ?R[owner-> ?DE],
    ?E.loc(?D) != ?R.location,
    timeRisk(?E,?T,?D,?TR),
    ?TR < 5.
```

Fig. 6. The second modification of the policy

As shown in in our example, defeasible reasoning can simplify the process of changing policies. Figure 7 shows the difference between the architectures of policies with and without using defeasible reasoning and object oriented logic programming. The architecture shown in Figure 7(b) is more modular than the one in Figure 7(a).



(a) Without defeasible reasoning and object orientation.



(b) With defeasible reasoning and object orientation.

Fig. 7. Possible architectures of policies

3.2 Virtual Hierarchies

In many cases, policy rules may conflict and be considered with regard to the position of the policy makers in the organizational hierarchy. For instance, suppose that policy makers x and y introduce policy rules $@r_x\ L_x \leftarrow Body_x$ and $@r_y\ L_y \leftarrow Body_y$ whose conclusions may conflict in some cases. If the organizational position of x is higher than y's, we can set the priority of rule r_x higher than that of r_y. There are two choices to apply such organizational hierarchies to policy rules: (1) the organizational hierarchy can be encoded in policy evaluator; or (2) we can use defeasible reasoning to allow policy rules of a lower-ranked actor to be defeated. Clearly, the second choice is more flexible than the first.

To represent organizational hierarchies of policy developers, we can assume that every rule in a policy is of the form $@r(x)\ L \leftarrow Body$ where x identifies the maker of the rule. We can represent the institutional hierarchy as a transitively closed set of facts of the form `boss(X,Y)` and then define the priorities of the policy rules as follows:

$$\backslash\mathtt{override}(\mathtt{r}(\mathtt{u_i}),\mathtt{r}(\mathtt{u_j})) :- \mathtt{boss}(\mathtt{u_i},\mathtt{u_j}). \tag{3}$$

4 Conclusion

In this paper, we argue that the use of defeasible reasoning can yield significant benefits in the area of role-based access control systems. As an illustration, we show that complex modifications to access control policies can be naturally represented in a logic programming framework with defeasible reasoning and they can be applied in modular fashion. The use of logic programming also easily supports various extensions such as institutional hierarchies. The same technique can be used to capture even more advanced features, such as distributed access control policies, Team-Based Access Control, and more.

There are several promising directions for future work. One is to investigate other access control models and, hopefully, accrue similar benefits. Other possible directions include incorporation of advanced features of object oriented logic programming, such as inheritance.

Acknowledgments. This work was supported, in part, by the Center for Dynamic Data Analysis (CDDA),[2] and NSF grant 0964196. We also thank Paul Fodor for his collaboration.

References

1. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. ACM Trans. Inf. Syst. Secur. **13**(3), 20:1–20:28 (2010). http://doi.acm.org/10.1145/1805974.1805976

[2] CDDA was supported by NSF award IIP1069147 and CA Technologies.

2. Brickley, D., Guha, R.: Rdf schema 1.1. Tech. rep., W3C (2014)
3. Chen, W., Kifer, M., Warren, D.S.: Hilog: A foundation for higher-order logic programming. The Journal of Logic Programming **15**(3), 187–230 (1993). http://www.sciencedirect.com/science/article/pii/074310669390039J
4. Damianou, N., Dulay, N., Lupu, E.C., Sloman, M.: The ponder policy specification language. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, p. 18. Springer, Heidelberg (2001). http://dl.acm.org/citation.cfm?id=646962.712108
5. Ferraiolo, D.F., Kuhn, R.D., Chandramouli, R.: Role-Based Access Control, 2nd edn. Artech House Inc, Norwood (2007)
6. Hitzler, P., Krtzsch, M., Parsia, B., Patel-Schneider, P.F., Rudolph, S.: Owl 2 web ontology language primer (second edition). Tech. rep., W3C (2012)
7. Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering DAC, MAC and RBAC. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 41–55. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-31540-4_4
8. Joshi, J., Bhatti, R., Bertino, E., Ghafoor, A.: An access control language for multi-domain environments. IEEE Internet Computing **8**(6), 40–50 (2004)
9. Kagal, L.: Rei1: A policy language for the me-centric project. Tech. rep., HP Laboratories (2002)
10. Kifer, M.: FLORA-2: An object-oriented knowledge base language. The FLORA-2 Web Site. http://flora.sourceforge.net
11. Kifer, M.: Rules and ontologies in F-logic. In: Eisinger, N., Małuszyński, J. (eds.) Reasoning Web. LNCS, vol. 3564, pp. 22–34. Springer, Heidelberg (2005)
12. Kifer, M., Lausen, G., Wu, J.: Logical foundations of object-oriented and frame-based languages. J. ACM **42**(4), 741–843 (1995). http://doi.acm.org/10.1145/210332.210335
13. Li, H., Zhang, X., Wu, H., Qu, Y.: Design and application of rule based access control policies. In: Proceedings of 7th Semantic Web and Policy Workshop (2005)
14. Parducci, B., Lockhart, H.: extensible access control markup language (xacml) version 3.0. Tech. rep., OASIS Standard (2013)
15. Park, J.S., Ahn, G.J., Sandhu, R.: Role-based access control on the web using ldap. In: Proceedings of the Fifteenth Annual Working Conference on Database and Application Security, Das 2001, pp. 19–30 Kluwer Academic Publishers, Norwell (2002). http://dl.acm.org/citation.cfm?id=863742.863745
16. Park, J.S., Sandhu, R., Ahn, G.J.: Role-based access control on the web. ACM Trans. Inf. Syst. Secur. **4**(1), 37–71 (2001). http://doi.acm.org/10.1145/383775.383777
17. Przymusinski, T.: Well-founded and stationary models of logic programs. Annals of Mathematics and Artificial Intelligence **12**(3–4), 141–187 (1994)
18. Schulzrinne, H., Tschofenig, H., Morris, J.B., Cuellar, J.R., Polk, J., Rosenberg, J.: Common policy: A document format for expressing privacy preferences. Internet RFC 4745, February, 2007
19. Sun, Y., Pan, P., Leung, H., Shi, B.: Ontology based hybrid access control for automatic interoperation. In: Xiao, B., Yang, L.T., Ma, J., Muller-Schloer, C., Hua, Y. (eds.) ATC 2007. LNCS, vol. 4610, pp. 323–332. Springer, Heidelberg (2007). http://dl.acm.org/citation.cfm?id=2394798.2394840

20. Wan, H., Grosof, B., Kifer, M., Fodor, P., Liang, S.: Logic programming with defaults and argumentation theories. In: Hill, P.M., Warren, D.S. (eds.) ICLP 2009. LNCS, vol. 5649, pp. 432–448. Springer, Heidelberg (2009)
21. Wan, H., Kifer, M., Grosof, B.: Defeasibility in answer set programs with defaults and argumentation rules. Semantic Web Journal (2014)
22. Yang, G., Kifer, M., Zhao, C.: FLORA-2: a rule-based knowledge representation and inference for the semantic web. In: Meersman, R., Schmidt, D.C. (eds.) CoopIS 2003, DOA 2003, and ODBASE 2003. LNCS, vol. 2888, pp. 671–688. Springer, Heidelberg (2003)