Int. J. Inf. Secur. (2017) 16:43–74
DOI 10.1007/s10207-016-0314-4

REGULAR CONTRIBUTION

A formal modeling and analysis approach for access control rules, policies, and their combinations

Vahid R. Karimi[1] · **Paulo S. C. Alencar**[1] · **Donald D. Cowan**[1]

Published online: 27 January 2016
© Springer-Verlag Berlin Heidelberg 2016

Abstract Approaches to access control (AC) policy languages, such as eXtensible access control markup language, do not provide a formal representation for specifying rule- and policy-combining algorithms or for verifying properties of AC policies. Some authors propose formal representations for these combining algorithms. However, the proposed models are not expressive enough to represent formally history-based classes of these algorithms, such as ordered-permit-overrides. In addition, some other authors propose a formal representation but do not present automated support for formal verification of properties of AC policies that use these algorithms. This paper demonstrates a new representation that can express all existing AC rule and policy combinations of which the authors are aware. This representation can also be used to automate the formal verification of properties of AC policies related to these algorithms. A new modeling representation for rule- and policy-combining algorithms based on state machines is used to specify rule- and policy-combining algorithms. Examples of these algorithms are programmed in the language of the SPIN model checker, and the programs are then used to support the automated formal verification of properties of AC policies. We present our approach and then use the AC policies and properties of CONTINUE, a conference management system, to compare it with prior work. Our first contribution is a new modeling representation for combining algorithms based on state machines. The second contribution is the formal verification of AC properties under certain combining algorithms that are beyond the capability of other approaches.

Keywords Access control (AC) · XACML · AC combining algorithms · Modeling · Analysis

1 Introduction

Access control constitutes an important component of operating systems, database management systems (DBMS), and applications. Access control policies define which users have access to what objects and operations and describe any constraints. Several incidents of information leaks in real systems owing to the implementation of incorrect access control policies have been reported (e.g., [10,13,58]). First, these occurrences point out the need for methodologies for modeling secure systems as several authors have advocated (e.g., [45]). Second, these incidents also indicate the need for a thorough analysis of access control policies and their properties. Although testing reveals many existing errors in software systems, errors still remain undetected even in safety-critical and economically vital systems [14].

An access control policy can be seen as a combination of one or more access control rules, and one or more policies can be combined into a set of access control policies that control access to an entire system. The rules and resulting policies can be combined in many different ways, and this combination can be achieved using the rule- and policy-combining algorithms of policy languages.

Approaches to access control (AC) policy languages, such as eXtensible access control markup language (XACML), do not provide a formal representation for specifying the rule-

✉ Vahid R. Karimi
vrkarimi@yahoo.ca

Paulo S. C. Alencar
palencar@uwaterloo.ca

Donald D. Cowan
dcowan@uwaterloo.ca

1 Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada

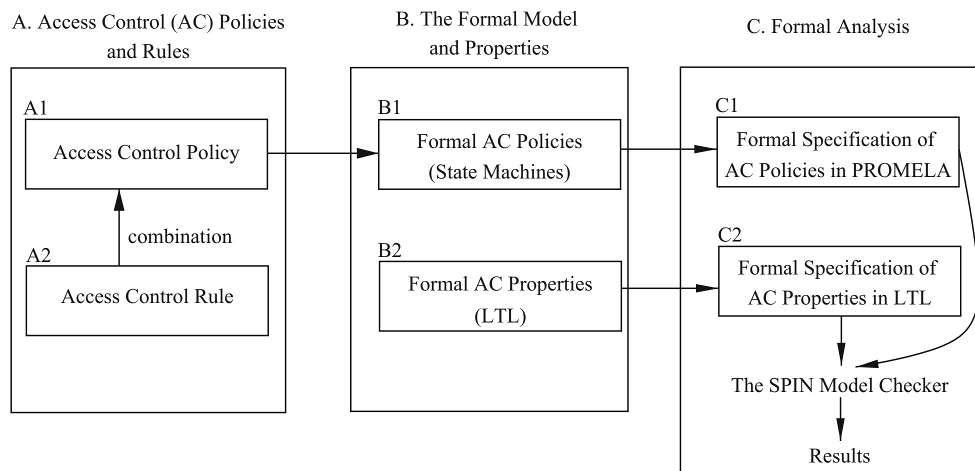

Fig. 1 A high-level overview of our approach

and policy-combining algorithms or for verifying properties of AC policies.

Some authors [18,36] propose formal representations for rule- and policy-combining algorithms. However, the proposed models are not expressive enough to represent formally classes of algorithms related to history of policy outcomes including ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable. In fact, they are not able to express formally any algorithm that involves history including the class related to consensus such as weak-consensus, weak-majority, strong-consensus, strong-majority, and super-majority-permit. In addition, some other authors, e.g., [38], propose a formal representation but do not present an approach and automated support for the formal verification of properties of AC policies in conjunction with the use of any classes of combining algorithms.

This paper presents a new modeling representation for access control rules, policies, and their combination and supports formal verification. This approach can express and verify formally all-known policy- and rule-combining algorithms, a result not seen in the literature. This approach supports automated formal verification, based on model checking, of single policies and combined policy sets.

Finally, the approach is applied to the AC policies and properties of CONTINUE, a well-known conference management system from the literature. Several properties, whose verification was not possible by prior approaches, such as ones involving history of policy outcomes, are verified, and results of this verification are shown in this paper.

2 An overview of our approach

Figure 1 shows an overview of the approach presented in this paper. A brief description of this figure is provided next.

2.1 Access control policies and rules

In its simplest form, an access control policy consists of a single access control rule, but normally several such rules are combined to make a policy as shown in Boxes A1 and A2 of Fig. 1. Similarly, several such policies can then be combined to make a policy set.

This paper uses general forms of state machines, represented in both algorithmic and diagrammatic forms to illustrate a rigorous systematic approach for describing all-known combining algorithms. Sections 3, 4, and 6 provide a detailed explanation of our approach.

2.2 The formal model and properties

Box B1 of Fig. 1 is the formalization of AC policies using AC rules and state machines where the AC rules govern the transitions between states.

The state machine representation supports the description and verification of more complex policies that are currently reported in the literature. For example, policies relying on history of policy outcomes and policies relying on consensus can be represented and verified in this formalism. This topic is discussed in Sects. 4 and 6.

Formal AC properties in linear temporal logic (LTL) are specified based on AC rules as shown in Box B2. This topic is presented in Sects. 5 and 6.

2.3 Formal analysis

The formal analysis of access control properties of policies is needed because access control policies can interact to produce undesirable behavior.

Box C1 shows the formal specification of AC policies. This step consists of the specification of state machines in the specification language of a model checker. Since this

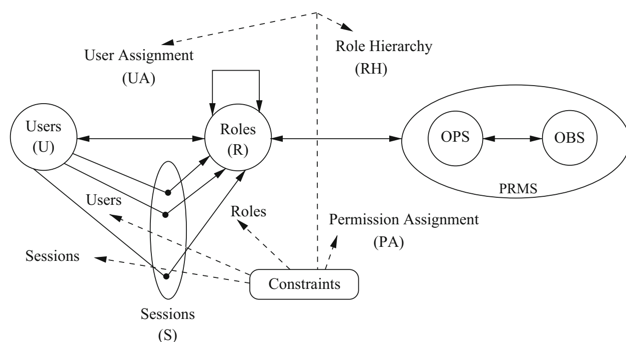

Fig. 2 The RBAC model

paper uses the SPIN (*S*imple *P*ROMELA *In*terpreter) model checker [8,23,24], state machines are encoded in PROMELA (*Pro*cess *Me*ta-*La*nguage), SPIN's C-like language.

We have decided to use the model checking approach in order to track the various states within the policies. In addition, various previous papers (see Sect. 7) have used model checking for the verification of access control policies. We chose SPIN, but other model checkers can be used. Box C2 portrays the formal specification of AC properties. In this step, properties are specified using the general form of properties from Box B2 and LTL that SPIN uses. Section 6 describes these two steps in detail.

3 Access control (AC), rules, and policies

Before describing AC rules and policies, a brief discussion on access control models is provided because one can view access control models as providing the basis for access control rules and policies. First, the role-based access control (RBAC) model [16,17,49], an example of a well-known and widely used access control model, is shown. After that, we describe the general concept of an access control model. Then, we provide the connection between an access control model, policy, and rule.

Figure 2 represents RBAC [16] in which permissions (PRMS) are shown as a many-to-many relationship between operations (OPS) and objects (OBS). According to Ferraiolo et al. [15], an access control model can be described by the five elements *users*, *objects*, *subjects*, *operations*, and *permissions*, and the relationships among them. A *user* represents an individual interacting with a computer system. An *object* represents any resource, such as a file, that can be accessed, and therefore is assumed to be passive. An *operation* is an active process such as write, when a user writes to a file, and a *subject* refers to a computer process, such as a program consisting of several *operations*.[1] Finally, a *permission* describes a set of tuples relating operations and objects such that if a tuple contains an operation and an object, then the operation on that object is permitted.

A mapping between these five elements and the *resources*, *events*, and *agents*, the elements used in this paper is described next. *Resources*, *events*, and *agents* are the components of a business modeling notation called resource–event–agent (REA) [20,26]. First, *users* and *objects* correspond to *agents* and *resources*, respectively. An *event* corresponds to the completion of an *operation* and can be considered as a different view of the same concept: an event includes a distinct change of state, whereas an operation makes the change happen [40]. Since a *permission* describes a set of tuples relating operations and objects, then based on the mapping between events and operations, permissions can also be modeled as tuples of events on resources. In addition, the subject corresponds to a process, which is set of operations that now map to *events*.

An AC rule can be expressed as an implication, which consists of a premise and a conclusion, as

$$p \rightarrow q \quad \text{or} \quad \textit{premise-rule} \rightarrow \textit{conclusion-rule}$$
$$\text{or} \quad \textit{p-rule} \rightarrow \textit{q-rule}$$

Instead of using p and q, this paper mainly uses *premise-rule* and *conclusion-rule* to be specific. *p-rule* and *q-rule* are used as a short form of *premise-rule* and *conclusion-rule* such as when state machines are presented to avoid clutter.

An example is provided next to indicate the general form of the premise and conclusion of an AC rule, to identify their elements, and to clarify the use of these elements in the state machines that will be presented shortly. This example and the rest of this paper use the terms *AgentType* (*AT*), *ResourceType* (*RT*), and *EventType* (*ET*) to describe the generic identification of an agent, resource, and event, respectively. The notion of type in REA is identical to that used by other authors (e.g., [42,43,52]).

The complete definition of the language is presented in detail in "Appendix 3."

Example 1 An AC rule in a banking system may be *tellers or managers are permitted to modify deposit accounts*.

In this example, the described *premise-rule* of the implication includes *tellers* or *managers* (i.e., roles), *deposit accounts* as *resources or objects*, and *modify* as an *event*. In addition, if these AC rules are based on a model (e.g., Fig. 2), then the existence of relationships, such as between operations (events) and objects (resources) or between roles and resources, is a part of the premise of an AC rule. In other

[1] The early access control models used the term *subject* for an active process, whereas in some recent descriptions of access control models, such as RBAC, an *operation* and a *subject* are distinguished [15] where a *subject* refers to a process possibly invoking several *operations*.

words, the *premise-rule* of this example includes the following elements:

(AgentType = Teller or AgentType = Manager)
and
(ResourceType = DepositAccount)
and
(EventType = Modify)
and
(RelATET(Teller, Modify) and
RelATET(Manager, Modify))
and
(RelRTET(DepositAccount, Modify)

RelATET(Teller, Modify) expresses a relationship between *Teller* and *Modify*, and RelATET(Manager, Modify) indicates a relationship between *Manager* and *Modify*. Similarly, RelRTET(DepositAccount, Modify) expresses a relationship between *DepositAccount* and *Modify*.

We define the *conclusion-rule*, of implication, as an EventResult expression that includes events and their results, i.e., *permit* and *deny*. One or more events are possible, and a *result* can be either a *permit* or *deny*. Therefore, the *conclusion-rule* (EventResult) for this example follows:

(Modify Access = permit)

A detailed description of the syntax of AC rules based on REA and using Extended Backus–Naur Form (EBNF) is provided in "Appendix 3." A complete presentation of rules in Backus–Naur Form (BNF) and EBNF can be found in the PhD thesis by Karimi [33].

The description provided for this example can be written in predicate logic as follows:

∀AgentType ∀ResourceType ∀EventType |
(AgentType = Teller ∨ AgentType = Manager)
∧
(ResourceType = DepositAccount)
∧
(EventType = Modify)
∧
(RelATET(Teller, Modify) ∧
RelATET(Manager, Modify))
∧
(RelRTET(DepositAccount, Modify))
→
ModifyAccess(permit)

4 AC policy and rule combinations

An AC policy set consists of a number of AC policies, and an AC policy is defined as a combination of one or more AC rules. As an AC policy usually consists of several rules, policy languages describe combining algorithms to provide different strategies for making decisions about this combination. For instance, a combining algorithm may permit a request if a rule in the collection of rules allows such a request, regardless of the existence or non-existence of another rule (within the collection) that denies such a request. Conversely, a combining algorithm may deny a request if one rule denies such a request and another rule within the collection permits the request. Two such algorithms for combining rules into policies are described in detail in Sect. 4.1.

This section (Sect. 4) presents the combination of AC rules and policies in two steps: first, an algorithmic form of state machines is shown to describe this combination (Sect. 4.1); this step corresponds to Box A1 of Fig. 1. Then, a diagrammatic representation of state machines is presented to describe the AC rule combinations (Sect. 4.2); this step corresponds to Box B1 of Fig. 1. Similarly, AC policy sets, which are combinations of AC policies, can be formed using the same state machine approach. Note that each state is either the premise or conclusion of an AC rule except for the initial and final states.

4.1 The use of algorithmic forms

One strategy for AC rule-combining is based on the first-applicable algorithm. The description of this algorithm follows [55,56]: the evaluation of rules within a policy is in the same order that rules are listed in a policy. If a rule applies, then the rule's result, i.e., *permit* or *deny*, applies and the evaluation of the rest of the rules halts. Otherwise, the procedure continues to the end. If none of the rules applies, then the result will be *not applicable*.

Figure 3 shows an algorithmic form for creating policies from rules for the first-applicable rule-combining algorithm. The description in this figure uses the notion of states and the premise conclusion of the AC rules as described in Sect. 3.

Another algorithm is permit-unless-deny [56], which can be described as follows: if any decision is deny, then the result will be deny; otherwise, the result will be permit.

Figure 4 shows an algorithmic form for the evaluation of rules for the permit-unless-deny rule-combining algorithm. Similarly, this description uses the premise and conclusion of AC rule definitions and also includes states.

4.2 The use of state machines

This section describes the use of formal state machines to represent algorithmic forms for combining rules into policies. Before showing these state machines, the associated states and their meanings are defined.

```
initial state = state q00;
for i = 1 to n do /* n = the number of rules in a policy */
    if premise-rule_i = false then
        move to state q_i0;
    else
        move to state q_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit;
            exit loop;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        move to state NA;
    end
end
```

Fig. 3 Rule combination using the AC rule definitions for the first-applicable algorithm

```
initial state = state q00;
for i = 1 to n do /* n = the number of rules in a policy */
    if premise-rule_i = false then
        move to state q_i0;
    else
        move to state q_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            continue;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        move to state permit;
    end
end
```

Fig. 4 Combining rules with the AC rule definitions for permit-unless-deny algorithm

State naming convention and meaning The initial state is q_{00}. With the exception of the initial state, the initial digit(s) of a state name is 1 or greater and indicate(s) the rule number. The last digit indicates whether the assumption of that rule holds (1) or does not hold (0); e.g., q_{11}: the state in which rule 1's assumption holds (i.e., true = 1).

q_{10}: the state in which rule 1's assumption does not hold (i.e., false = 0).

As previously mentioned, each rule on state machines is represented as *p-rule* $\rightarrow$ *q-rule*, where *q-rule* consists of the EventResult part.

The assumptions and conclusions for rules 1 to n can be shown as $p\text{-}rule_1 \ldots p\text{-}rule_n$, and $q\text{-}rule_1 \ldots q\text{-}rule_n$, respectively. Each assumption consists of one or more atomic statements that are combined using conjunction, disjunction, or negation. Similarly, each conclusion consists of one or more atomic statements that are joined by conjunction, disjunction, and negation.

Similarly, when the state machine is used to describe policy combinations instead of rule combinations, the word *rule* changes to *policy* in the description. For instance, in a policy-combining state machine, q_{11} and q_{10} have the following meanings:

q_{11}: the state in which policy 1's has the outcome in which the assumption holds (i.e., true = 1).
q_{10}: the state in which policy 1's has the outcome in which the assumption does not hold (i.e., false = 0).

Transition within state meaning The transitions between states are governed by examining the *p*-rule and *q*-rule. As mentioned previously, the *p*-rules on state machines are identical to the assumption part of a rule. The conclusion of a rule or EvenResult on state machines is shown as follows:

$$\left[\bigwedge_{i=1}^{m_1} \text{Event}_i \text{Access (permit)} = \text{true}\right]$$

The superscript of $\bigwedge_{i=1}^{m_1}$ indicates the number of elements (i.e., events and their results) in the EventResult expression of a rule. For instance, m_1 is the number of elements in the EventResult of rule 1, and m_2 is the number of elements in the EventResult of rule 2.

Figure 5 shows the UML state machine for the first-applicable algorithm.

Figure 6 shows the UML state machine for the permit-unless-deny rule-combining algorithm.

4.3 Policy-combining algorithms

Policy-combining algorithms are similar to the rule-combining algorithms. For instance, the first-applicable policy-combining algorithm can be described as follows [55,56]: the evaluation of policies within a policy set is in the same order that policies are listed in a policy set. If a policy applies and its result is *permit* or *deny*, then the result (i.e., *permit* or *deny*) applies and the evaluation of the rest of the policies halts. If a policy does not apply or its result is *not applicable*, then the procedure continues. If no other policy exists, then the result will be *not applicable*.

Figure 7 shows an algorithmic form for describing the first-applicable policy-combining algorithm. Figure 8 shows

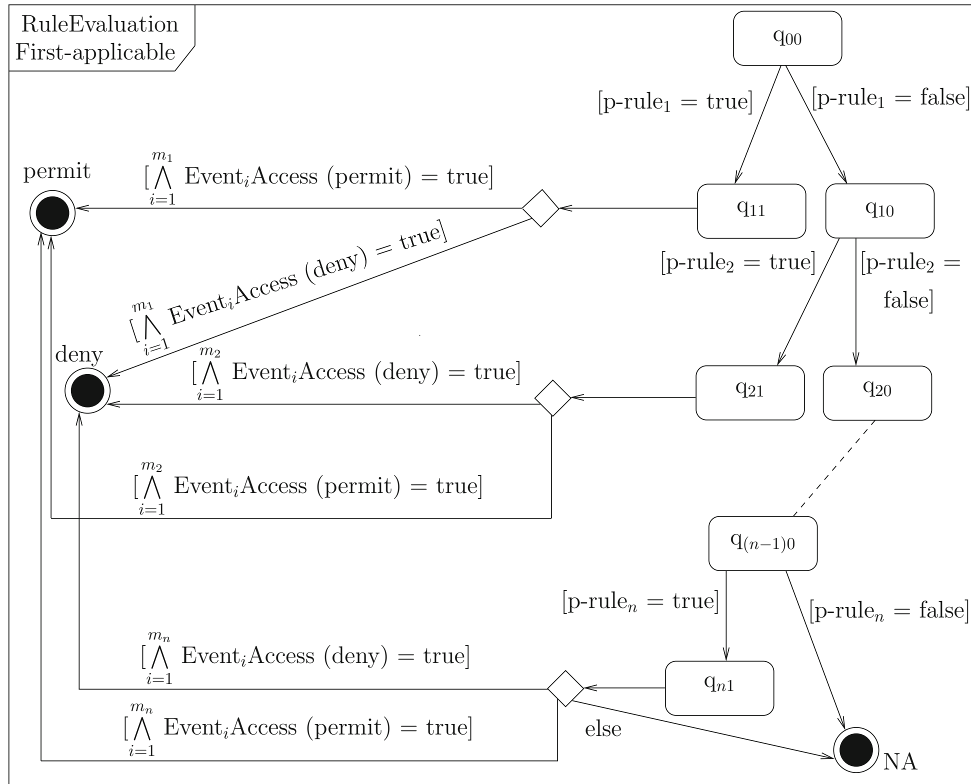

Fig. 5 A UML state machine using AC rule definitions for the first-applicable algorithm

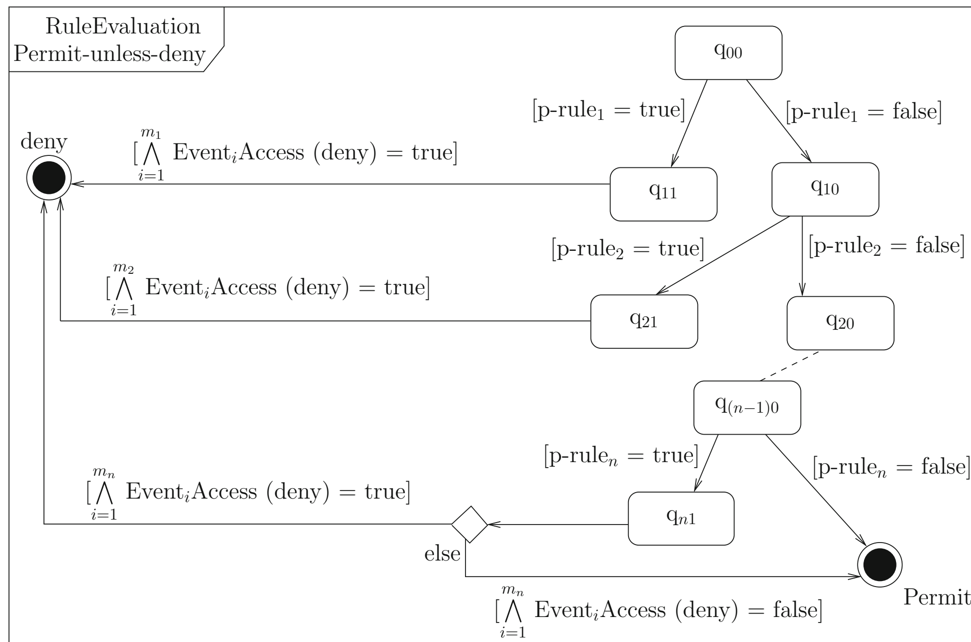

Fig. 6 A UML state machine that uses the definitions of AC rules for the permit-unless-deny algorithm

```
initial state = state q00;
for i = 1 to n do                          /* n = the number of policies in a policy set */
    if premise-policy_i = false then
        move to state q_i0;
    else
        move to state q_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit;
            exit loop;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny;
            exit loop;
        end
    end
    if i = n then
        move to state NA;
    end
end
```

Fig. 7 An algorithmic description for the first-applicable policy-combining algorithm

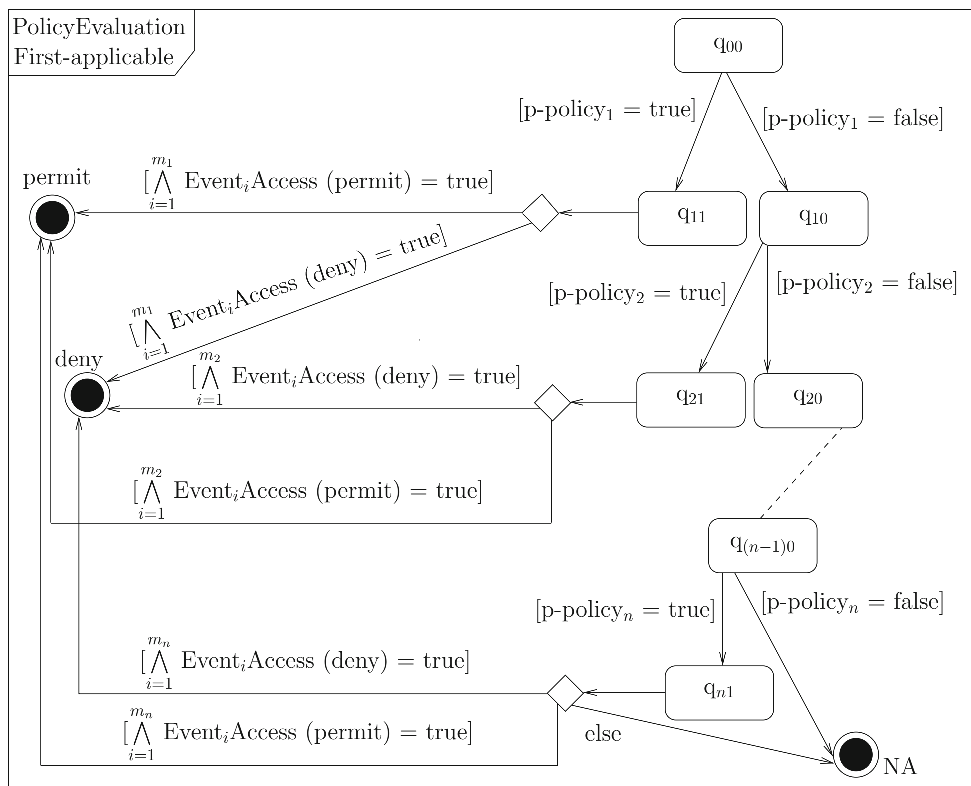

Fig. 8 A UML state machine for the first-applicable policy-combining algorithm

its corresponding state machine for the first-applicable policy-combining algorithm.

4.4 A key advantage of our approach

Policy languages such as XACML can express the combination of rules and policies. Li et al. [38] illustrate the approach with examples such as weak-consensus. XACML can describe this combination in the form of pseudo-code, but we provide algorithmic forms that are comparable with the pseudo-code format; in addition, we also provide accompanying state machine descriptions.

In comparison, our approach using state machines allows us to describe rule and policy combinations and to analyze them through model checking.

We emphasize the formality of our approach by describing weak-consensus using our notation.

Weak-consensus [38] "Sub-policies should not conflict with each other: Permit a request if some sub-policies permit a request, and no sub-policy denies it. Deny a request if some sub-policies deny a request, and no sub-policy permits it. Yield a value indicating conflict if some permit and some deny."

Figure 9 shows the state machine representation for the weak-consensus policy-combining algorithm that corresponds to the algorithmic form, which is provided in Fig. 19, "Appendix 2."

Other possible (combining algorithms) approaches, suggested by Li et al. [38], such as the strong-consensus policy-combining algorithm, weak-majority policy-combining algorithms, strong-majority policy-combining algorithm, and super-majority-permit policy-combining algorithm are very similar and can be described using the same approach. For instance, as another example, the weak-majority policy-combining algorithm is shown in "Appendix 2."

5 Formal properties

Before describing a general form of AC property, background information on temporal implications and translations from predicates to propositions is provided.

Temporal implications This paper uses temporal implications as discussed by Holzmann [23]. Table 1 provides a summary of temporal implications.

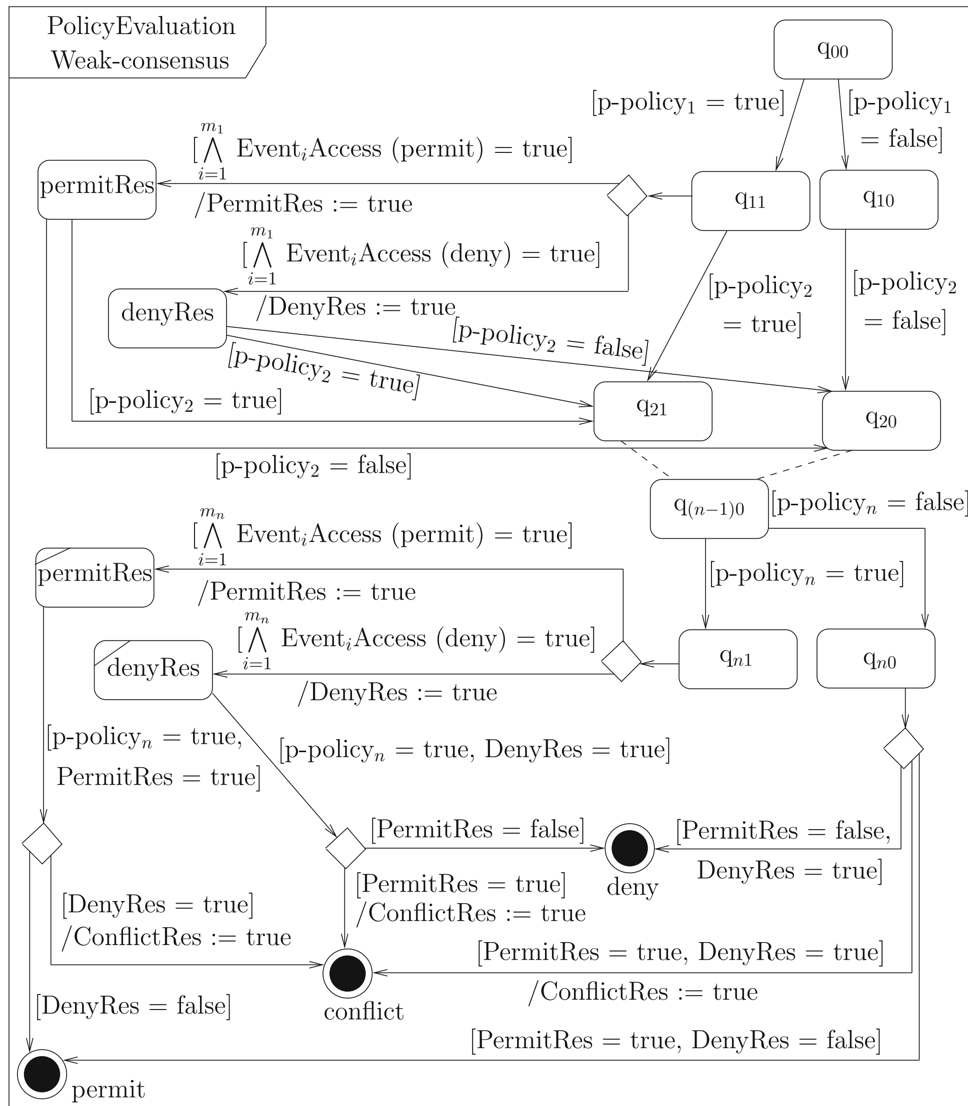

Fig. 9 A UML state machine representing the weak-consensus policy-combining algorithm

Table 1 Temporal implications [23]

To describe p implies q, one can write a simple expression, $p \rightarrow q$
As $p \rightarrow q$ is equivalent to $(!p) \vee q$, this implication holds, for instance, in the first state of a run in which p is false or q is true. In order to make the implication true within each step of a run, the implication must be written as $\Box(p \rightarrow q)$, where $\Box$ means always.
This expression is still not correct because it has no notion of temporal implication (e.g., the evaluation of a rule reaches a result at some point). This expression must be written as $\Box(p \rightarrow \Diamond q)$, where $\Diamond$ means eventually.
The latest expression still holds in a case in which p and q hold at the same state. In order to capture the notion that the truth of q somehow is caused by the truth of p, one can change the description by adding the next operator (X) to have $\Box(p \rightarrow X\Diamond q)$
Finally, the last expression holds if p never becomes true because of the implication ($\rightarrow$). The addition of $\wedge\Diamond p$ at the end of the previous expression ensures that p is expected to hold at some point of time. This addition prevents the expression from being vacuously true. The final description is $\Box(p \rightarrow X\Diamond q) \wedge\Diamond p$

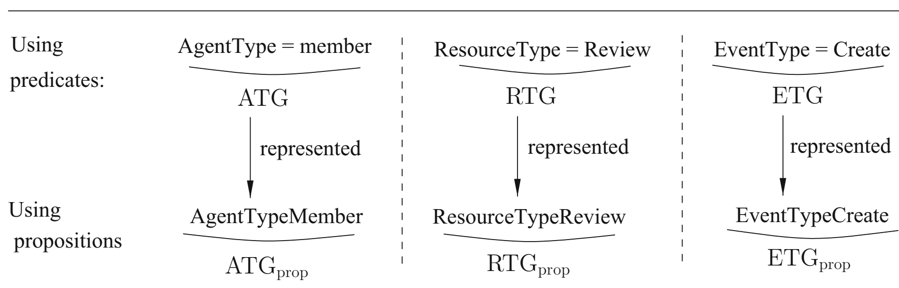

Fig. 10 Predicate and propositional versions of expressions

From predicates to propositions Properties are specified in LTL, which is a propositional temporal logic. First, Figs. 10 and 11 provide the general idea for the translation from predicates, such as equalities and relationships, to propositions. The propositional versions are identified with a subscript of prop in these figures. Then, PROMELA, the language of SPIN, is used to describe this translation at the code level. Figure 10 shows three elements ATG, RTG, and ETG and their propositional versions identified as $\mathrm{ATG}_{\mathrm{prop}}$, $\mathrm{RTG}_{\mathrm{prop}}$, and $\mathrm{ETG}_{\mathrm{prop}}$, respectively. The complete definition of the language in detail, which includes the definitions for terms ATG, RTG, ETG among others, is provided in "Appendix 3."

ATG, RTG, and ETG are the minimum identification of Agent, Resource, and Event, respectively. For instance, ATG can only include the name of agents, but AgentExp can include the agent's attribute names and their values. Similar explanations hold for RTG and ETG in relation to their counterpart ResourceExp and EventExp. As previously mentioned, this paper and Fig. 10 use terms AgentType (AT), ResourceType (RT), and EventType(ET) to describe the generic identification of agent, resource, and event, respectively.

The propositional elements of Fig. 10 can be described by the language of a model checker. For instance, if PROMELA is used, then the propositional versions in this figure (i.e., AgentTypeMember, ResourceTypeReview, and EventTypeCreate) can be defined as follows.

In the following expressions, the symbol "==" means equal in PROMELA. Therefore, for instance, the first line of the following expression defines an element AgentTypeMember to be the propositional equivalent of the predicate version of the expression (AgentType == member) and holds the same information as one unit.

A similar explanation holds for the second and third definitions: ResourceTypeReview and EventTypeCreate are the propositional versions of the expressions (ResourceType == Review) and (EventType == Create), respectively.

```
#define AgentTypeMember (AgentType == Member)
#define ResourceTypeReview (ResourceType ==
                                        Review)
#define EventTypeCreate (EventType == Create)
```

Similarly, predicates, such as the ones that represent relationships, can be described as one-unit-propositions as shown in Fig. 11.

The CONTINUE case study that we shortly describe is written in the RBAC profile of XACML. There is a relationship between roles and the operations element of permissions in RBAC, and another relationship between operations and objects of permissions. We included these relationships in PROMELA. We also mention more about our decision to include relationships in Sect. 6.8. The predicates that can represent existing relationships, such as between agent types and event types ATET(Member, Create), can be represented by the language of a model checker. For instance, using PROMELA, this relationship can be expressed using an array called RelA, which is defined by the keyword *typedef*. As shown next, the elements of this array are AgeN and Act,

which can hold information about agent types and event types. This relationship can be expressed as an array in which one field (i.e., AgeN) represents agent types, and the other field (i.e., Act) exemplifies event types. The typedef keyword is used to declare a user-defined data structure. In this example, we define an array in which each index of this array has two fields.

```
typedef RelA {
              byte AgeN;
              byte Act;
             }
```

Then, a specific array of the required size can be defined, e.g., `RelA memR[1]`, one can define the size of this array to be equal to the number of relationships between agent types and event types.

For instance, the elements of this specific array can hold information about a relationship between an agent type of *member* (where the AgeN field identifies *member*), and an event type of *create* (where the Act field exemplifies *create*) as follows. In the following expressions, the symbol "=" is for assignment.

```
memR[0].AgeN = member; memR[0].Act = create;
```

Similar to the previously provided description to represent predicates by propositions, the propositional version of the relationship between an agent type of *member* and an event type of *create* can be represented using PROMELA's keyword *define*, as shown next. ETMemberCreate is the propositional representation of the information within the array just described. In other words, the following expression defines an element ETMemberCreate to be the propositional equivalent of the predicate version of the expression (memR[0].AT == member && memR[0].ET == create) and holds the same information as a single unit in one proposition.

```
#define ETMemberCreate (memR[0].AT == member
   && memR[0].ET == create)
```

The predicates, such as CreateAccess(permit) or its equivalent representation in the form, *Create Access = permit*, can be defined as CreateAccess == Permit. The latter expression can be defined as a proposition called CreateAccesPermit, as shown next.

```
#define CreateAccessPermit (CreateAccess ==
                                    Permit)
```

5.1 General form of AC property specification

Figure 12 shows a general form of an AC property. This definition includes the temporal operators of LTL and the connectives of propositional logic.

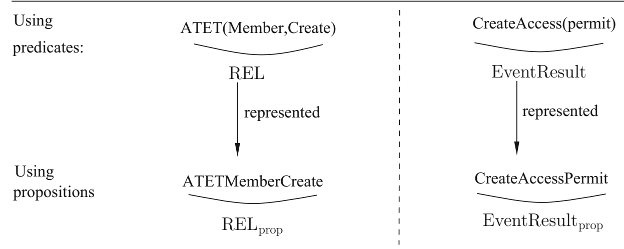

Fig. 11 Propositional versions of predicates

P = uop (P) | (P bop P) | ultl (P) | (P bltl P) | AgentExp_{prop} | $\text{ResourceExp}_{prop}$ | EventExp_{prop} | AgeEveRel_{prop} | ResEveRel_{prop} | $\text{EventResult}_{prop}$ | ATG_{prop} | RTG_{prop} | ETG_{prop}
uop = "not";
bop = "and" | "or" | "implies";
ultl = "always" | "eventually" | "next";
bltl = "until" | "release" | "weak until";

Fig. 12 A general form of AC Property

An atomic proposition is represented as "p." AgentExp_{prop}, $\text{ResourceExp}_{prop}$, EventExp_{prop}, AgeEveRel_{prop}, ResEveRel_{prop}, $\text{EventResult}_{prop}$, ATG_{prop}, RTG_{prop}, and ETG_{prop} are propositions equivalent to AgentExp, ResourceExp, EventExp, AgeEveRel, ResEveRel, EventResult, ATG, RTG, and ETG, respectively.

As described previously (Sect. 5), the elements ATG_{prop}, RTG_{prop}, and ETG_{prop} are the minimum descriptions of AgentExp_{prop}, $\text{ResourceExp}_{prop}$, and EventExp_{prop}, respectively.

For instance, the element ATG_{prop} is the propositional description of an agent's name, whereas the element AgentExp_{prop} is a propositional description that can also include an agent's attribute names and their values. Similar explanations hold for RTG_{prop} and ETG_{prop} in relation to their counterparts $\text{ResourceExp}_{prop}$ and EventExp_{prop}.

One specific example of this general form is provided in Sect. 6.6, in which we discuss the specification of properties in LTL.

6 CONTINUE conference management case study

This section shows the application of the materials on access control models, rules, and policies, and their combinations from Sects. 3 and 4 (corresponding to Boxes A1, A2, and B1 of Fig. 1) to build access control policies in CONTINUE, a conference management system. This section also elaborates the approach of encoding CONTINUE's access control policies using PROMELA (corresponding to Box C1 of Fig. 1).

CONTINUE's policy and property descriptions use the approach of Sects. 3, 4, and 5 and are described as shown

in Box C of Fig. 1. The properties are specified in linear temporal logic (LTL) and then verified using the SPIN model checker.

Properties of policies that use the first-applicable combining algorithm are then verified, and the results are expressed and compared with the outcomes of two papers by Fisler et al. [18] and Kolovski et al. [36] that use the same case study but apply different verification techniques. The approach is similar to these two papers in that the first-applicable combining algorithm is used, and the verification time and state space, obtained by using SPIN, are provided. Then, three combining algorithms—ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable—that involve history of policy outcomes are described using state machines and the approach of this paper. These algorithms are not capable of being described, for the purpose of formal verification of properties, using any other approach presented in the current literature. The ordered-permit-overrides algorithm is then described in full including the verification results of properties.

6.1 CONTINUE, policies, and properties

CONTINUE [37] is a free conference management application supporting the submission, review, discussion, and notification phases of conferences. A broad description of CONTINUE's behavior follows:

- During the initial stage, individuals can view the conference information.
- During the submission phase, authors, including program committee (PC) members, but not PC chairs, can submit papers.
- PC chairs assign papers to non-conflicted PC members (i.e., PC members cannot be assigned their own papers to review).
- Only those who are assigned papers to review can submit reviews.
- No PC members can view other PC members' reviews unless the former have submitted their own reviews. The purpose of preventing PC members from accidentally accessing other member reviews, before submitting their own, is to reduce bias in their reviews.
- PC chairs can see all decisions, but PC members do not have this authorization. PC members who have submitted papers should not be able to determine who reviewed their papers.
- Paper reviews are read during the discussion phase and are the basis for decisions on which papers are accepted.

The original conference management access control policies are described using eXtensible access control markup language (XACML). The CONTINUE policies are available in XACML format on the CONTINUE Web site.[2] For brevity, a prose description of the XACML policies is provided in "Appendix 1."

The XACML format and also the prose translation demonstrate the difficulty of defining access control policies correctly because of the numerous rules and their nested referrals. Furthermore, CONTINUE describes properties that are provided later in this paper. This paper specifies properties and uses the general form shown in Sect. 5.

6.2 AC rule combination by algorithmic form and state machine

The approach shown in this paper can describe various ordering combination algorithms, such as ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable for the purpose of formal verification, whereas prior works are not capable of expressing these combinations. These three combining algorithms and their specifications are described next based on the approach provided in this paper.

The ordered-permit-overrides rule-combining algorithm This algorithm can be described as follows [55,56]: The evaluation of rules within a policy is in the same order that these rules are listed in a policy. If any rule evaluates to *permit*, then the result is *permit*. If none of the rules evaluates to *permit* and the result of at least one evaluation is *deny* and the results of the rest are *not applicable*, then the result is *deny*. If none of the rules applies, then the final evaluation result is *not applicable*.

Figure 13 shows in an algorithmic form a description of ordered-permit-overrides.

Figure 14 shows the UML state machine corresponding to the ordered-permit-overrides rule-combining algorithm. Permit-overrides, another algorithm, is similar to ordered-permit-overrides with one difference: in permit-overrides, rules can be evaluated in any order within a policy. This algorithm can be implemented using the non-deterministic if-statement of SPIN. Guards (options or choices) within an if-statement or a loop can be non-disjoint; as a result, one of them can be selected non-deterministically. This selection can even be different from one execution to the next.

The ordered-deny-overrides rule-combining algorithm This algorithm can be described as follows [55,56]: The evaluation of rules within a policy is in the same order that these rules are listed in a policy. If any rule evaluates to *deny*, then the result is *deny*. If none of the rules evaluates to *deny* and the result of at least one evaluation is *permit* and the results of the rest are *not applicable*, then the result is *permit*. If

[2] http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/.

```
set seen-deny to false; initial state = state q00;
for i = 1 to n do /* n = the number of rules in a
policy */
    if premise-rule_i = false then
        move to state q_i0;
    else
        move to state q_i1;
        if Event_i Access(permit) = true for every
        element of EventResult then
            move to state permit; exit loop;
        else if Event_i Access(deny) = true for every
        element of EventResult then
            move to state deny-seen;
            set seen-deny to true;
        end
    end
    if i = n then
        if seen-deny = true then
            move to deny state;
        else
            move to state NA;
        end
    end
end
```

Fig. 13 The defined AC rules and states for ordered-permit-overrides

none of the rules applies, then the final evaluation result is *not applicable*.

Figures 15 and 16 show the algorithmic form representation and the UML state machine for the ordered-deny-overrides algorithm, respectively.

Similar to the previous case, another rule-combining algorithm is deny-overrides in which rules can be evaluated in any order within a policy. Similarly, the use of the non-deterministic if-statement of SPIN, as previously explained, applies in this case too.

The only-one-applicable rule-combining algorithm the XACML standard defines the only-one-applicable combining algorithm for policies not rules.

Kolovski et al. [36] describe the only-one-applicable rule-combining algorithm because the same rule- and policy-combining algorithms of XACML are similar. This algorithm can be described as follows: if more than one rule is applicable, then the result is *indeterminate*. If none of the rules is applicable, then the result is *not applicable*. If only one rule applies, then the result of that rule applies. The only-one-applicable rule-combining algorithm in the algorithmic form

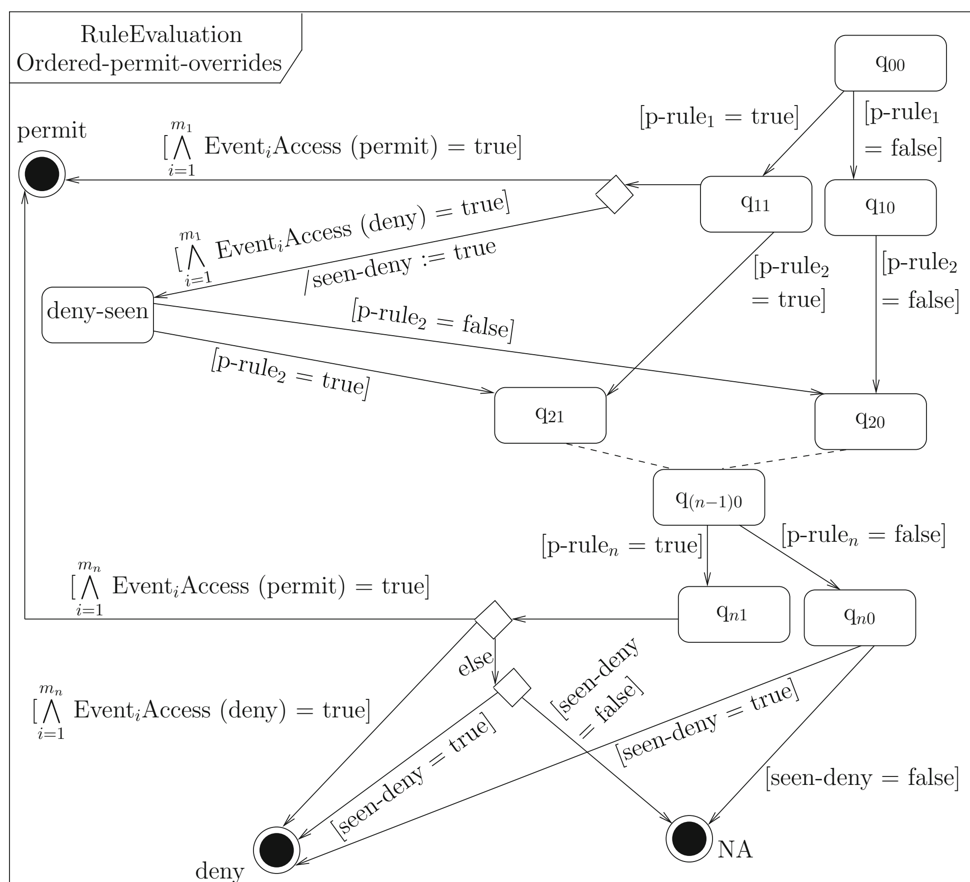

Fig. 14 A UML state machine using the defined AC rules for ordered-permit-overrides

```
set seen-permit to false;
initial state = state q00;
for i = 1 to n do                                  /* n = the number of rules in a policy */
    if premise-rule_i = false then
        move to state q_i0;
    else
        move to state q_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            move to state permit-seen; set seen-permit to true;
        else if Event_i Access(deny) = true for every element of EventResult then
            move to state deny; exit loop;
        end
    end
    if i = n then
        if seen-permit = true then
            move to state permit;
        else
            move to state NA;
        end
    end
end
```

Fig. 15 The defined AC rules and states for ordered-deny-overrides

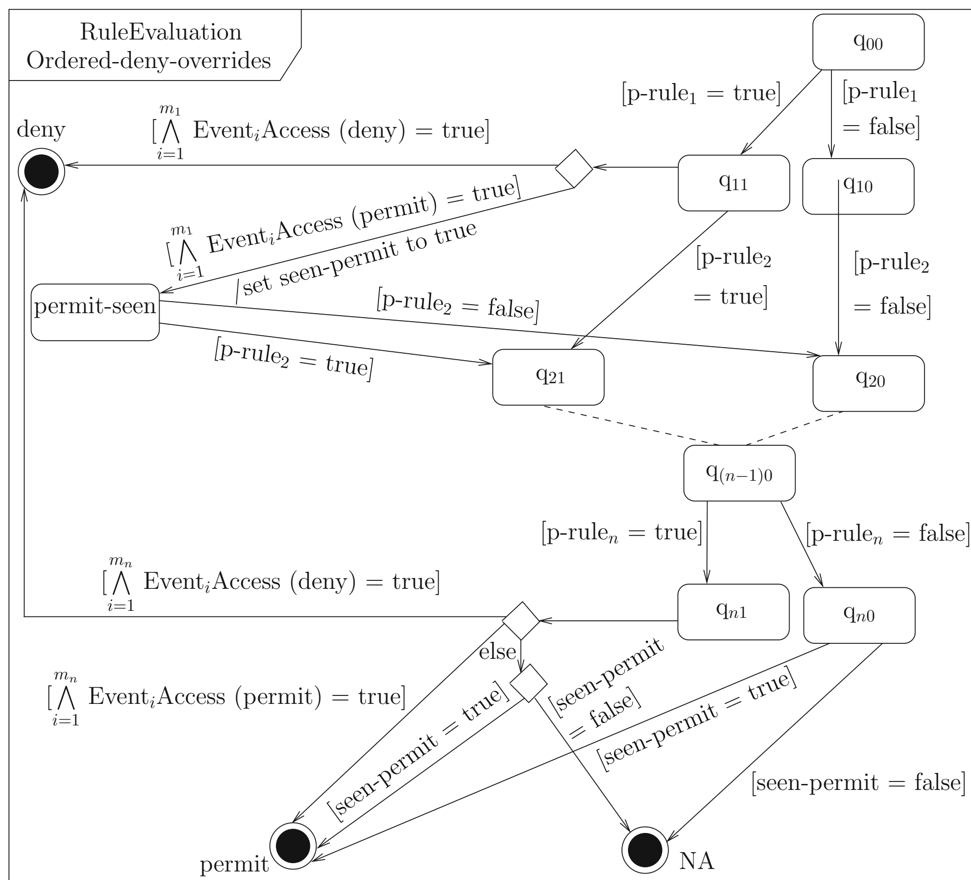

Fig. 16 A UML state machine using the defined AC rules for ordered-deny-overrides

```
initial state = state q00;
set num-seen to zero;
set result to false;
for i = 1 to n do                                    /* n = the number of rules in a policy */
    if premise-rule_i = false then
        move to state q_i0 ;
        if i = n and num-seen > 1 then
            move to state indeterminate;
            exit loop;
        else if i = n and num-seen = 0 then
            move to state NA;
            exit loop;
        else if i = n and num-seen = 1 and result = permit then
            move to state permit;
            exit loop;
        else if i = n and num-seen = 1 and result = deny then
            move to state deny;
            exit loop;
        end
    else
        move to state q_i1;
        if Event_i Access(permit) = true for every element of EventResult then
            set result to permit;
            add one to num-seen; move to state seen;
        else if Event_i Access(deny) = true for every element of EventResult then
            set result to deny;
            add one to num-seen; move to state seen;
        end
        if i = n and num-seen > 1 then
            move to state indeterminate;
            exit loop;
        else if i = n and num-seen = 1 and result = permit then
            move to state permit;
            exit loop;
        else if i = n and num-seen = 1 and result = deny then
            move to state deny;
            exit loop;
        end
    end
end
```

Fig. 17 The defined AC rules and states for only-one-applicable

of this paper is shown in Fig. 17, and Fig. 18 is the corresponding state machine.

6.3 An advantage of this paper's approach

Table 2 shows a summary and comparison of this paper with prior approaches in terms of using rule-combining algorithms in the context of formal verification of properties. The plus sign indicates the capability of the methods of Fisler et al. [18] and Kolovski et al. [36] in formal description and verification, whereas the minus sign shows the inability of their methods to express a formal description and subsequent verification.

Similarly, an expression of the combination of ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable rule-combining algorithms with others is not possible by other approaches. As a result, if a policy uses the first-applicable algorithm, and another applies ordered-permit-overrides, then their combination cannot be expressed using the approaches of either Fisler et al. or Kolovski et al. Table 3 shows the various possible combinations.

6.4 Formal analysis

This section uses the SPIN model checker to express CONTINUE's policies and properties in addition to the analysis of properties. SPIN accepts specifications written in a C-like language called PROMELA. Correctness properties can be written as assertion statements or specified as LTL formulas.

Access control was initially defined using a matrix to specify who has access to what, but the growth of an organization adds to the maintenance problems of such a matrix; instead, access control is now defined using rules to describe the infor-

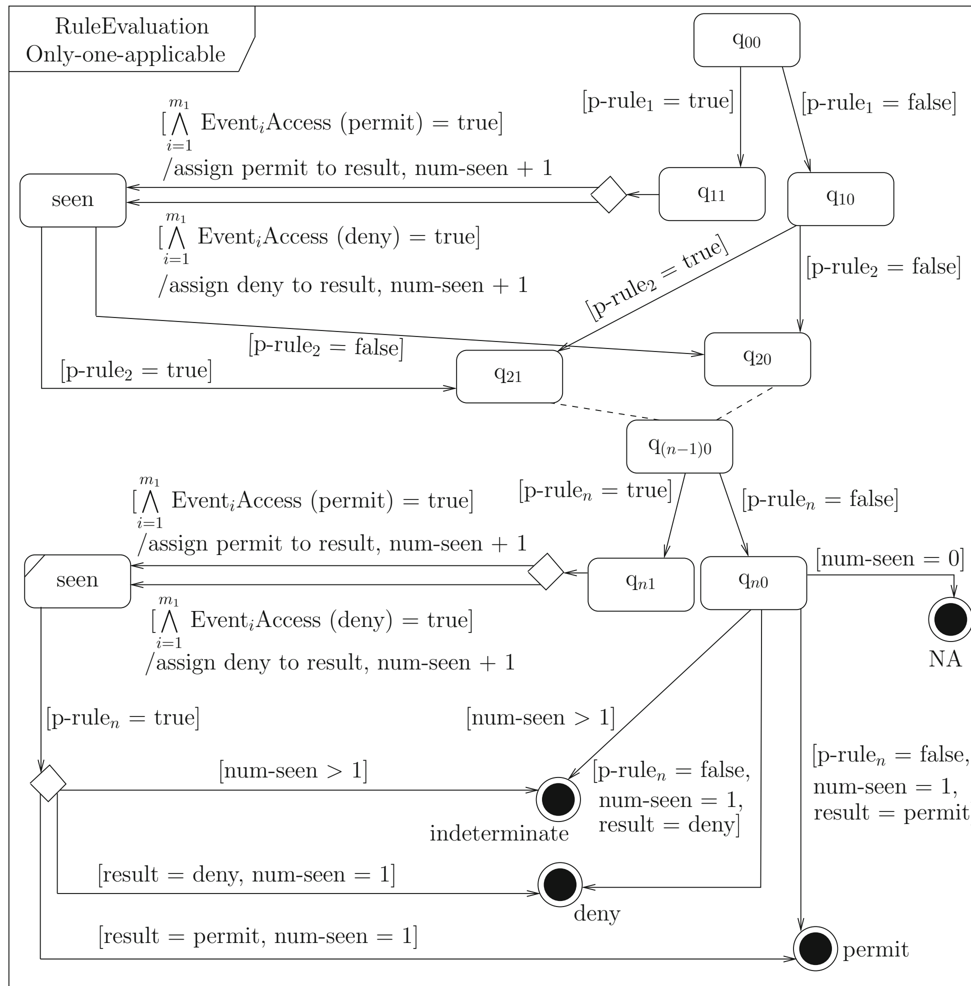

Fig. 18 A UML state machine using the defined AC rules for only-one-applicable

Table 2 Rule-combining algorithms in the context of their use with formal verification

Rule-combining algorithms	Fisler et al. [18]	Kolovski et al. [36]	This paper
First-applicable	+	+	+
Permit-overrides	+	+	+
Deny-overrides	+	+	+
Ordered-permit-overrides	−	−	+
Ordered-deny-overrides	−	−	+
Only-one-applicable	−	−	+

mation within that matrix [18]. Despite the benefits of using rules, some disadvantages still exist [35]: (1) large organizations have a lot of rules; therefore, application of these rules makes it problematic to determine who has access to what, and (2) the addition and modification of new rules add to the problem of maintaining these rules. (Although the disadvantages are described only in the context of rule-based RBAC, they apply generally). In addition, it is well known that what one specifies is what one gets but not necessarily what one wants. Therefore, the analysis of specifications still constitutes a significant step no matter how carefully a specification is described.

6.5 Formal specification of AC policies in PROMELA

The existing CONTINUE policies, described in XACML, are specified in PROMELA. The formats of AC rules have also been previously defined. These rules are encoded in PROMELA.

Table 3 Rule-combining algorithms in the context of using with formal verification

Rule-combining algorithms	Fisler et al. [18]	Kolovski et al. [36]	This paper
First-applicable + ordered-permit-overrides	–	–	+
First-applicable + ordered-deny-overrides	–	–	+
First-applicable + only-one-applicable	–	–	+
Permit-overrides + ordered-permit-overrides	–	–	+
Permit-overrides + ordered-deny-overrides	–	–	+
Permit-overrides + only-one-applicable	–	–	+
Deny-overrides + ordered-permit-overrides	–	–	+
Deny-overrides + ordered-deny-overrides	–	–	+
Deny-overrides + only-one-applicable	–	–	+

In addition, this specification of AC rules follows the algorithmic forms or state machines to encode policies and their combinations. The specific AC rules of CONTINUE are specified in their place-holders, identified as rule-numbers, in algorithmic forms or state machines.

The following example shows one possible combination of policies. The following if-statement shows the premise of rules, such as premiseRule1 and premiseRule2, and the conclusion of rules, such as conclusionRule1 and conclusionRule2. The premises and conclusions of rules are defined in PROMELA as shown in the top portion of Fig. 10, using equalities and arrays as described on p. 10. Initially, the state is q_{00}, as shown in the state machines of this paper. If premiseRule1 holds, then the state is q_{11}, and based on conclusionRule1, the state proceeds to a result (*permit* or *deny*). Otherwise, the state is state q_{10}, and the procedure continues to the state representing premiserule2. The procedure uses the same approach in evaluating the policy to the end.

```
if
::  premiseRule1 -> conclusionRule1;
::  else
    if
    :: premiseRule2 -> conclusionRule2;
    :: else
       if
       :: premiseRule3 -> conclusionRule3;
       fi;
    fi;
fi;
```

6.6 Formal specification of AC properties in LTL

CONTINUE has properties that can be expressed in LTL. The SPIN model checker is used to specify properties. Properties can be specified as LTL expressions in SPIN. Table 4 shows the mathematical symbols and the SPIN equivalents used.

Table 4 Some LTL and SPIN operators

Operator	LTL	SPIN
And	$\wedge$	&&
Or	$\vee$	||
Not	$\neg$	!
Implies	$\rightarrow$	->
Always	$\Box$	[]
Eventually	$\Diamond$	<>
Next	X	X

Table 5 provides the CONTINUE properties.

Property Example For any state, if an individual is neither a PC chair nor an administrator, then he or she cannot eventually set (write) the meeting flag resource.

This property can be defined in the LTL notation of SPIN as

```
[](pThree -> X<>notSetup)
```

where `pThree` and `notSetup` are defined as follows:

```
#define pThree (!(AT == chair || AT ==
                admin) && (RT ==
                ismeetingflagR) &&
                eventIsW && chaERrel
                && admERrel)
```

where `eventIsW` is defined as

```
#define eventIsW (ET == writeEvent)
```

The purpose of `chaERrel` is to define the existence of relationships between the role of *chair* and the operation of *write* and between the operation of *write* and the resource *ismeetingflagR*. We have described the same concept in Sect. 5 describing *typdef* in PROMELA. For simplicity in that section, we have included two fields to represent the relationship between agent types and event types. We can include a third field as a resource to represent the relationship between operation and resource. We discuss more about the inclusion of relationships in Sect. 6.8. `chaERrel` is defined as

Table 5 Properties

The CONTINUE properties are provided next. Note that because of the use of model checking and temporal logic and implications (Table 1), *always*, *eventually*, and *next* are used in the description of these properties.

Property 1 ($\mathbf{Pr}_1$): For any state, if there is a request, then it will eventually allow only a deny or permit response (i.e., no NA response is possible). This property should hold.

Property 2 ($\mathbf{Pr}_2$): It is always the case that a program committee (PC) member who owns reviews can eventually edit his/her reviews. This property should hold.

Property 3 ($\mathbf{Pr}_3$): For any state, if an individual is neither a PC chair nor an administrator, then he or she cannot eventually set the meeting flag. In verification, this property should hold.

Property 4 ($\mathbf{Pr}_4$): It is always the case that if an individual role is not described (i.e., no roles exist for a subject), then no permit exists for the individual. In verification, this property should fail.

Property 5 ($\mathbf{Pr}_5$): For any state, if an individual's role is not described and a resource is not conference information, then eventually no permits exist for the individual. This property should hold.

Property 6 ($\mathbf{Pr}_6$): It is always the case that if a person is neither a PC chair nor an administrator, then the person should eventually never be allowed to read the *paper-review* resources for which he/she has a conflict of interest. This property should hold.

Property 7 ($\mathbf{Pr}_7$): For any state, if an individual is neither a PC chair nor an administrator, and he/she is conflicted, then the individual should never be eventually permitted to read either any part of the *review-content-set* resources that are not written by the individual, or read the reviewer-info resource. This property should hold.

Pr_7 and Pr_6 are similar in the sense that the latter refers to a subset of the former. (Pr_6 refers to *paper-review* resources, and Pr_7 includes all review resources). A discrepancy between Pr_7's prose description and the specification of this property in Scheme exists. The specification in Scheme considers only review-content-set and paper-review-info-reviewer resources, not all resources; nevertheless, either case is acceptable.

Property 8 ($\mathbf{Pr}_8$): For any state, if a PC chair calls for a meeting, then the chair can eventually read anything related to the subject of the meeting. This property should fail.

This property possibly exists to check whether a paper's certain information, such as a paper's author(s), can be revealed in a meeting. Generally, the specific information, which can be known, must be clearly defined.

Property 9 ($\mathbf{Pr}_9$): For any state, if a PC chair calls for a meeting, then the chair can eventually read any part of the reviews to be discussed at the meeting. This property should hold.

Pr_9, a specific case of Pr_8, allows access to all eight review resources (described previously).

Property 10 ($\mathbf{Pr}_{10}$): It is always the case that a non-conflicted PC member at discussion phase can eventually read all parts of the reviews. This property should hold.

Property 11 ($\mathbf{Pr}_{11}$): For any state, a non-conflicted PC member who has submitted a review of a paper can eventually read all parts of others' reviews of that paper. This property should hold.

For both Pr_{11} and Pr_{12}, the provided Scheme code on the CONTINUE Web site specifies non-conflicted PC members.

Property 12 ($\mathbf{Pr}_{12}$): It is always the case that when the phase is not discussion, a PC member who has been assigned to submit a review of a paper but who has not done so cannot eventually read any part of *review-content-set* of others' reviews of that paper. This property should hold.

```
#define chaERrel (chaR[13].AgeN == cha &&
                  chaR[13].WAct == writeA &&
              chaR[13].ResN == ismeetingflagR)
```

Similarly, the purpose of `admERrel` is to define the existence of relationships between the role of *administrator* and the operation of *write* and between the operation of *write* and the resource *ismeetingflagR*. `admERrel` is defined as

```
#define admERrel (admR[0].AgeN == adm &&
                  admR[0].WAct == writeA
                  && admR[0].ResN ==
                  ismeetingflagR)
```

`notSetup` is defined as

```
#define notSetup (writeAccess == deny)
```

We could have used the term writeDenied instead of notSetup to be more descriptive but decided to use the latter term to be consistent with the description of the property (i.e., set the meeting flag).

This property is a specific example of the general form of properties provided in Fig. 12, Sect. 5.1. This specific example in connection with the general form can be described as follows:

(i) `pThree` is a propositional description and includes several elements.
(ii) The element, `AT == chair`, corresponds to ATG, and its propositional expression within `pThree` corresponds to $\mathrm{ATG}_{\mathrm{prop}}$. Similarly, the element, `AT == admin`, is another use of ATG.

(iii) The “or” (||) term corresponds to one option of *bop* in Fig. 12.
(iv) The expression, `RT == ismeetingflagR`, corresponds to RTG, and its propositional expression within `pThree` corresponds to RTG_{prop}.
(v) The “and” (`&&`) term is one option of *bop* of the general form.
(vi) The element, `eventIsW`, is ETG_{prop} of the general form. This element is the propositional expression and corresponds to the predicate, `ET == writeEvent`.
(vii) The two elements, `chaERrel` and `admERrel`, are examples of the description of general forms $\text{AgeEveRel}_{\text{prop}}$ and $\text{ResEveRel}_{\text{prop}}$.
(viii) The symbol implication (`->`) is one option of *bop* of the general form.
(ix) Finally, the specific definition, `notSetup`, is an example of the general form of $\text{EventResult}_{\text{prop}}$.

6.7 Verification results and expressive advantage

This paper reports on the verification of the twelve properties provided by CONTINUE. The result of this verification is described and compared with the results of two previous papers. These two papers use different approaches that are not completely comparable with the experiments described in this paper, but they use CONTINUE policies and properties and use verification as a part of their efforts. The approach in this paper to verify access control policies using model checking is completely comparable with the work described by Jha et al. [29] as they also use a model checker.

The state space of a program is the multiplication of the number of statements of each process by the number of values each variable can have [8]. The state space in this work was relatively small because the range of values that each variable could take has been selected to be as small as possible. In addition, the number of statements has been reduced by using PROMELA’s *atomic* keyword.

Table 6 shows the experimental results for the properties that hold. The first row of data is the state space; the second row is the verification time in seconds, and the third is the memory usage reported when running the program. Similarly, the next three rows represent state space, running time, and memory usage for the same properties, but the assumptions of implications that must eventually hold are not included. For instance, based on the explanation provided in Table 1, for Pr_1, the top three rows report the experiment for the expression `[](p− > X <> q) && <> p`, and the bottom three rows of column Pr_1 use the expression `[](p− > X <> q)`, an acceptable form, in which the assumptions that must eventually hold are not stated.

Finally, Pr_4 and Pr_8 failed as they should. Pr_4 failed with the state space of 293,315, a running time 0.654 s, and a memory usage of 101.036 MB. Pr_8 failed with the state space of 16,313, a running time of 0.047 s, and a memory usage of 7.969 MB. A Windows PC with a Pentium (R) Dual core 2.7 GHz CPU and 4 Gbytes of memory is used for this experiment.

As an example, the CONTINUE case study is used again with the ordered-permit-overrides rule-combining algorithm. All 12 properties whose verifications under this algorithm were not possible by prior approaches are verified. Table 7 shows the result of this verification.

Pr_4 and Pr_8 failed as they should. Pr_4 failed with the state space of 274,906, a running time 0.623 s, and a memory usage of 94.883 MB. Pr_8 failed with the state space of 16,313, a running time of 0.044 s, and a memory usage of 7.969 MB. Similar to the previous case, a Windows PC with a Pentium (R) Dual core 2.7 GHz CPU and 4 Gbytes of memory is used for the experiment.

It is worth mentioning that the computational limitations imposed by the model checking approach apply to our work. One main limitation of model checking is state space explosion, which implies that as the model gets larger, the analysis will be impractical. Another limitation of model checking is that the analysis is performed on a model of a system and not on the actual system [5]. This limitation is similar to one open problem pointed out by Fisler et al. [19] about access control policy analysis. They mention that there is a need for “tractable models and analysis techniques for the interactions between policies and the software systems that use them. Most policy analysis papers ignore the surrounding software system, thus losing valuable information.”

Table 6 State space (total number of states), verification time (in s), and memory usage (in MB) with first-applicable rule-combining algorithm

	Pr_1	Pr_2	Pr_3	Pr_5	Pr_6	Pr_7	Pr_9	Pr_{10}	Pr_{11}	Pr_{12}
States	1,364,017	496,401	538,257	368,017	368,017	436,497	383,633	438,033	387,089	381,953
Time	12.3	3.06	3.2	2.14	2.14	2.7	2.31	2.74	2.37	2.23
Memory	468.298	170.426	184.796	126.349	126.349	149.859	131.710	150.387	132.896	131.133
States	700,017	380,049	377,233	368,017	368,017	377,745	370,833	372,433	369,521	368,465
Time	3.75	1.91	1.84	1.89	1.85	1.93	1.92	1.93	1.8	1.88
Memory	240.332	130.479	129.513	126.349	126.349	129.688	127.315	127.865	126.865	126.502

A comparison of our experiment and those of two others is described next. This comparison is not in terms of state space and verification time directly because the other two experiments use different approaches. One reports a certain timing for parsing XACML and constraining the representation, and the other describes timing for parsing XACML, converting the representation to description logic, and preprocessing time to convert them to normal form. Neither of these elements applies to the work described in our approach.

Next, two other experiments that use the same policies and properties but apply different verification methods are discussed. These works are by (a) Fisler et al. [18] and (b) Kolovski et al. [36].

The Work by Fisler et al. [18] The authors, who created and made CONTINUE policies and properties publicly available, use multi-terminal binary decision diagrams (MTBDDs) to represent policies. MTBDDs, variations of binary decision diagrams, have multiple terminal nodes such that the *permit*, *deny*, and *not applicable* of a policy rule can be represented. An MTBDD is built for each rule; then, these MTBDDs are combined. MTBDDs can be manipulated using PLT Scheme (renamed Racket), a programming language also used to query and verify properties. The authors report a time of 2050 ms to parse and convert policies into the MTBDD representations and another 20 ms to constrain this representation. The verification of each property takes <1 ms. They obtained these results using a machine with an Athlon XP 1800+ processor at 1.5 GHz with 512 MB RAM.

The verification time of Fisler et al.'s experiment is faster than those results reported in Table 6, but Fisler et al.'s 2050 ms for parsing policies and 20 ms for constraining policies are not applicable and have not occurred in the experiment using SPIN because compiling the PROMELA program is fast. Fisler et al. use PLT Scheme to write queries about policies, whereas LTL expressions are used in our approach.

Fisler et al.'s experiment and ours use different machines, and we have provided the specification of these machines. Fisler et al. provide a change impact analysis and a tool that translates XACML policies into a venue for their analysis, but we do not. We explain our approach for the inclusion of a change impact analysis as future work in Sect. 6.8. Although we do not provide a tool for translating XACML policies into PROMELA, this paper includes an extensive use of state machine modeling that can serve as a blueprint for such a tool. We provide further explanations on this subject, such as the inclusion of relationships, in Sect. 6.8. On the other hand, Fisler et al. cannot describe any of ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable combining algorithms, but the approach provided in this paper can explicitly represent such descriptions and perform verifications on the policies that use such combining algorithms.

The Work by Kolovski et al. [36] The authors used description logics and implemented a prototype of an XACML analysis tool on top of Pellet, a description logic reasoner. In general, description logics are decidable subsets of first-order logic, but some are supersets of predicate logic. The authors mention that one advantage of choosing description logic representations is their expressiveness. Therefore, a larger subset of XACML that is more expressive than propositional logic can be represented and verified. Parsing these XACML policies took 2.1 s, and converting them to description logic took another 1.7 s. Preprocessing concepts and transforming them into normal forms consumed 10.6 s. Verification of properties took 0.420 s on average. The type of machine used to obtain these results is not mentioned. The authors attribute the faster time reported by Fisler et al.'s work to the optimizations for Pellet that are designed to perform verification for a richer logic than propositional logic (i.e., the logic that describes these policies).

Kolovski et al.'s 0.42 s verification time is faster than the one reported in Table 6, but their 2.1 s for parsing, 1.7 s for converting policies to description logic, and 10.6 s for transforming concepts to normal forms are not applicable to the experiment using SPIN because the compilation of the PROMELA program is fast. (Note that Tables 6 and 7 show the running time).

Kolovski et al.'s experiment and ours use different machines. Kolovski et al. provide a change impact analysis and a tool that translates XACML policies into a venue

Table 7 State space (total number of states), verification time (in s), and memory usage (in megabytes) with ordered-permit-overrides rule-combining algorithm

	Pr_1	Pr_2	Pr_3	Pr_5	Pr_6	Pr_7	Pr_9	Pr_{10}	Pr_{11}	Pr_{12}
States	1,391,409	503,249	545,105	374,865	374,865	450,833	390,481	444,881	393,937	388,801
Time	12.8	3.17	3.33	2.21	2.21	3.02	2.4	2.87	2.4	2.32
Memory	477.702	172.777	187.147	128.700	128.700	154.781	134.61	152.738	135.248	133.484
States	713,713	386,897	384,081	374,865	374,865	387,089	377,681	379,281	376,369	375,313
Time	3.81	1.95	1.95	1.87	1.9	1.96	1.93	1.94	1.91	1.93
Memory	245.034	132.031	131.864	128.700	128.700	132.896	129.666	130.216	129.216	128.853

for their analysis, but we do not. As formerly expressed, we explain our approach for the inclusion of a change impact analysis as future work in Sect. 6.8. As mentioned previously, we do not present a tool for translating XACML policies into PROMELA, but this paper accommodates a large number of state machine models that can assist as a blueprint for such a tool. We present more explanations on this topic, such as the inclusion of relationships, in Sect. 6.8. On the other hand, Kolovski et al. [36] are not able to specify ordered-permit-overrides and ordered-deny-overrides combining algorithms, but our approach can. Kolovski et al. believe that they will be able to express only-one-applicable combining algorithm in their future work.

6.8 Discussion and future work

This section provides a discussion of the running time of the case study using different approaches in addition to the inclusion of relationships and its consequence for the running time of case study. Furthermore, the presentation covers absolute guarantee of scalability, future work, the use of model checking, and other possible approaches.

First, neither our work nor the other experiments (Kolovski et al. [36], Hughes and Bultan [27], Arkoudas et al. [3], the latter two papers are described in the Sect. 7) that used the initial case study produced a faster running time than Fisler et al.'s approach. The purpose of these other experiments and approaches is to illustrate feasibility and additional capabilities. This paper also proposes a feasible approach with the additional feature that our work can describe and verify some combining algorithms that were not previously possible.

We have mentioned in Sect. 5 about the inclusion of relationships. The CONTINUE case study has been written as an RBAC profile in XACML. We have decided to write access control rules based explicitly on the access control model. As a result, in this case (i.e., RBAC) there is a relationship between a role and an operation, and another one between an operation and an object, and we have included these relationships. Of course including the relationships makes the size of our case study larger than other referenced work and increases the running time.

In addition, there is another important element to consider. If the goal is to obtain an absolute guarantee of scalability, then that goal cannot be achieved no matter how many experiments are performed. Hughes and Bultan [27] explain this notion well "All we can say is that our approach performs efficiently on these examples, and can successfully verify non-trivial properties on these policies. Assessing the effectiveness of our verification approach in practice would require a comprehensive study, which is beyond the scope of this work." It should be noted that the largest example that Hughes and Bultan used is CONTINUE. Hughes and Bultan state "Since the examples we tested so far were easily handled by the SAT solver we believe that our approach will be feasible for analysis of large XACML policies." We agree with these statements, and they represent our view of our experiments. One aspect of our future work is to conduct more experiments. Furthermore, another direction of our future work consists of inclusion of impact analysis and other types of analysis as a verification formulation.

Finally, we conclude this section with a discussion about model checking use and our choice to apply this approach. Model checking has previously been used for the verification of access control policies (see Sect. 7). We agree with the statement, which follows, by Jha et al. [29] (see Sect. 7) who also use model checking for the analysis of access control policies. They state "In real-world large-scale RBAC systems, even though the number of roles in the whole system may be large, we expect that the roles that are relevant for any given query will be only a small portion of all roles." Similarly, we believe that the number of variants such as roles in a query is a small portion of all possibilities. Furthermore, the inclusion of combining algorithms causes selective paths of the program to be activated, and therefore, the state space of each run will be a small portion of all possibilities. Thus, we expect the model checking approach will be scalable in this sense. In our experiments, we have used SPIN, an explicit-state model checker.

We now explore several topics about model checkers. These topics, discussed next, are the use of modularization and its applicability related to our work, industrial use of model checkers, current and future enhancements of model checkers, and the use of advanced features of model checkers. There may always be concerns that model checkers may not scale as the number of policies grows. One remedy is to divide policies into different modules and perform verification accordingly. Our work is related to XACML policies, and a certain number of policies are specified in one XACML file. Several XACML files can exist with their combining algorithms. Even for a very large number of policies, it will be unlikely that all policies are specified in one XACML file. Therefore, the model checker does not need to explore the verification of policies within one verification. This modularization process of XACML can work in favor of scalability of verification. In other words, if needed, certain number of policies can be combined and verified together using their combining algorithms, and then this process can continue.

Despite the state space explosion issue, model checkers are used in industry. The SPIN Web site lists success stories such as the application of SPIN in the automotive industry. Here SPIN and its Swarm verification front-end is applied where Swarm "performs many small verification jobs in parallel, that can increase the problem coverage for very large verification problems."[3]

[3] http://spinroot.com/spin/success.html.

Furthermore, model checkers implement optimization techniques. SPIN uses the *partial order reduction* technique to reduce the number of reachable states that must be searched to verify properties. To reduce state space, SPIN also utilizes *statement merging* in which "sequences of transitions within the same process" are combined into a single step [23]. SPIN also supports a *slicing* algorithm that determines based on given properties which statements can be omitted from the model without effecting the verification of those properties [23]. SPIN likewise implements (complementary techniques to state space reduction) strategies to reduce the amount of memory needed to store each state. Two such strategies are *collapse* compression and *minimized automaton* (MA) compression that are elaborated in depth by Holzmann [23].

Because of these and other enhancements to model checking, the variety and size of the problems that can be handled has increased substantially. Two additions to the SPIN model checker are the use of parallelism and the addition of the Swarm tool. These additions take advantage of production of multi-core CPUs [25]. The parallel search algorithm has been introduced in SPIN in 2005 by a modification (that has been lately enhanced) enabling "the execution of the depth-first search analysis on multiple cpu-cores" [22]. Search optimization through the Swarm tool can be used when a large number of CPUs or CPU-cores are available. A user provides the tool with three parameters: the number of CPUs or CPU-cores, the size of memory available for a run, and the upper limit of runtime for the search to be completed. The tool enables verification without exceeding the memory and time limits specified by the user.

Even though model checkers include advanced features to optimize models and reduce state spaces, users may not use these features. They should be used for very large verification problems. For instance, features such as *slicing* and *memory compression* must be invoked in SPIN to take effect. Running the slicing algorithm displays possible redundancies in the model for a stated property. Moreover, by using the option—DCOLLAPSE—when compiling a program, a user can collapse the size of state vectors (i.e., the amount of memory required to encode a single state) up to 80–90 % [48].

Furthermore, one may postulate that if a more general language (instead of the use of model checkers) is used for expressing combining algorithms, then any combining algorithms such as the ordered versions of combining algorithms can be expressed. We clarify that the idea is not whether a language or an approach is expressive enough to present a feature, but whether the approach (language or tool) is designed to express that feature. The model checking approach is designed to keep track of states, and for this reason, we have chosen our approach that also enables us to express ordered versions of combining algorithms. For instance, Alloy is based on predicate logic that is more expressive than the logic that Fisler et al. [18] use, but as Fisler et al. [18] and Hughes and Bultan [27] explain, these authors were not capable of handling the analysis of CONTINUE properties when they used Alloy. The conclusion that can be made is that at least at that time Alloy was not designed to solve the type of problems that Fisler et al. and Hughes and Bultan intended to solve. This inability to solve that type of problems is not relevant to the expressivity of Alloy.

7 Related work

Several policy languages have been suggested in the literature. XACML [55,56] is the standard XML access control policy language. XACML describes rule- and policy-combining algorithms in English and provides pseudo-code for the application of these algorithms. In contrast, this paper uses algorithmic forms and state machines where the AC rules govern the transitions between states. In addition, this paper provides the translation of state machines to the language of the SPIN model checker and performs the formal verification of properties of access control policies, which is beyond the scope of XACML.

The enterprise privacy authorization language (EPAL) [4] represents another policy language. Despite some similarities between EPAL and XACML, differences exist between these two policy languages [1]: EPAL does not support the inclusion of one policy within another one, and therefore unlike XACML, EPAL does not provide a language to define the results from several policies. In addition, a policy in EPAL can have several rules, and if the effect of the first rule is applicable, the subsequent rules within a policy are ignored. Therefore, the approach shown for the first-applicable combining algorithm in this paper can be modified to apply to EPAL because EPAL does not support policy-combining algorithms and allows only a form of first-applicable rule-combining algorithm. Ni and Bertino [47] suggest eXtensible functional language for access control (xfACL). One goal of xfACL is to strengthen XACML. A policy evaluation in this language can have one of the following results [47]: permit, deny, null (no decision), not applicable (NA), either permit or not applicable (PNA), either deny or not applicable (DNA), either permit or deny (PD), either permit, deny, or not applicable (PDNA). The authors plan to extend their work to include policy analysis techniques.

Fisler et al.'s work [18] and Kolovski et al.'s work [36] have been described in detail in Sect. 6.7. As previously mentioned, neither one is capable of describing some rule-combining algorithms, such as ordered-permit-overrides and ordered-deny-overrides, for the purpose of formal verification.

Two related research papers by Hughes and Bultan [27] and Arkoudas et al. [3] use the same conference manage-

ment case study and use formal verification techniques to analyze XACML policies. For this reason, we discuss them in detail. Nevertheless, these two research approaches do not discuss ordered versions of combining algorithms. Both of these papers provide change impact analysis and a tool to translate XACML policies for analysis.

Hughes and Bultan [27] translate queries for XACML policies into Boolean satisfiability problems and use a SAT solver to perform verification. They explain the translation of XACML policies into their approach and provide the description for first-applicable, permit-overrides, deny-overrides, and only-one-applicable combining algorithms.

Hughes and Bultan compare their work with the work by Fisler et al. [18] with an experiment and show their results for eleven properties. To obtain the time for verifying each property within their experiment, four different components need to be added together. These components are I/O processing, transformation to triple form, Boolean formula generation and conjunctive normal form (CNF) transformation, and SAT solving where the first and third components dominate the total analysis time. The authors state that the comparison of their work with the work by Fisler et al. is difficult because the architecture of the two approaches is different. In Fisler et al.'s approach, XACML policies are parsed into a form suitable for analysis once, and then all properties are verified. In this process, the parsing time is dominant, and the verification time is very fast. Hughes and Bultan note if the time for parsing and verification of properties of Fisler et al.'s experiment are added together, then this total is comparable with the total obtained by adding the four components in their proposal. However, for Fisler et al.'s experiment, the parsing is performed only once followed by the verification of properties, whereas for Hughes and Bultan's work the addition of four components must be done for each property. Their experiment is performed using a machine with 2.8 GHz Intel Pentium 4 and 2 GB of memory. The authors conclude that by using their approach, it is feasible to verify XACML policies. Similarly, we have shown that our approach makes it possible to verify XACML policies.

Hughes and Bultan state that their approach is capable of expressing only-one-applicable combining algorithm that is not possible by Fisler et al.'s research. Nevertheless, they do not express the representation of any ordered versions of combining algorithms such as ordered-permit-overrides or ordered-deny-overrides. We have expressed the presentation of only-one-applicable algorithm in addition to expressing ordered-permit-overrides and ordered-deny-overrides.

Arkoudas et al. [3] use satisfiability modulo theories (SMT) [44], which are described as a generalization of propositional satisfiability, to analyze policies. In addition to the use of an SMT solver, another essential component of their work is Athena [2] that is a functional programming language (with some imperative features) and an interactive theorem prover based on many-sorted first-order logic. Athena interacts with an SMT solver through a procedure called *smtSolve*. The SMT solver that they used is Yices.

Arkoudas et al. use their approach to load CONTINUE policies into their tool and verify the twelve properties of CONTINUE. The authors express a one time loading time of about 1.4 s and a total verification time of 570 ms for all properties. Loading is in three phases. For their experiment, Arkoudas et al. use a machine with 2.53 GHz Intel Core Duo CPU with 2.96 GB of RAM. The authors also report analyzing synthetically generated policies with 10–1000 rules that have from 4–200 attribute values. For this analysis, a combining algorithm of first-applicable, permit-overrides, deny-overrides, or only-one-applicable was randomly chosen. The authors indicate that the structure of their polices can be atomic or complex where the latter corresponds to XACML policies. In contrast, our work is only in connection with XACML policies. Arkoudas et al. indicate their approach is capable of expressing combining algorithms first-applicable, permit-overrides, deny-overrides, and only-one applicable in addition to expressing any complex combinations; nevertheless, they do not express any work or results in terms of ordered versions of these combining algorithms.

Arkoudas et al. explain that their approach is capable of expressing a wide range of policy analysis such as *consistency* (i.e., a request is not both permitted and denied by a policy) and *conflict* (e.g., one policy creates a different decision from another policy for the same request). As the authors attest, most of their range of policy analysis can be expressed as verification problems (i.e., whether a given policy satisfies certain properties). One direction that our future work can take is to investigate expressing this range of policy analysis as property verification.

We have used a model checker in this paper to analyze access control policies. Model checking has previously been used for the verification of access control policies. We discuss a few representative papers illustrating this line of research. Jha et al. [29] use model checking and logic programming to analyze access control policies and compare these two approaches. They perform two experiments and conclude that logic programming using XSB (the Stony Brook University Extended Prolog) performs better for small instances, while model checking using the new symbolic model verifier (NuSMV) performs better for larger cases. The authors' work provides insights in terms of using two different analysis approaches, but the scope of their work is limited to RBAC. Their work is not concerned with rule- and policy-combining algorithms that exist in policy languages. Schaad et al. [50] use a model checker to verify delegation and revocation functionality. They use the NuSMV model checker in the context of a real-world banking workflow to confirm unexpected use of delegation and revocation functionality that can violate separation of duties. Similar to our approach,

they use a model checker to analyze access control properties, but their work is not about XACML policies and combining algorithms. Zhang et al. [59] describe a model checking algorithm for analyzing access control policies written in a language based on propositional logic. This language is called RW (where R stands for *reading*, and W denotes *writing*). In RW, a property or a query is a group of agents and a goal is either reading or writing data. The authors use a small example related to an *employee information system* to show their approach that can be used in two modes. The *assessing mode* is whether a property holds, and the *intrusion detection mode* is concerned with what steps can be taken to make a property achievable if a property does not hold. Our work is related to their first mode (i.e., assessing mode), and they are not concerned with combining algorithms.

Other formal verification techniques are used to specify and analyze access control policies. For instance, several authors use Alloy [28] to specify and analyze aspects of access control and usually use small examples to illustrate their approach. Schaad and Moffett [51] use an illustrative example that is specified and analyzed in Alloy to demonstrate conflicts that can arise when role-based access control and delegation are used. The authors describe conflicts that can occur regarding delegation and separation of duties. Toahchoodee and Ray [53] examine the policy integration using Alloy and state this integration is not trivial. The authors use a small example and an algebra in which an authorization consists of a triple of sets of *subjects*, *objects*, and *actions*. They show how to describe the union, intersection, and subtraction, of two policies while storing the result in a third policy using Alloy. The authors also discuss the closure of a policy using a derivation rule. Mankai and Logrippo [39] discuss conflicts among access control policies and use Alloy for analysis. The authors describe an XACML access control policy that consists of three rules about reading and modifying grades. Their example is translated into Alloy, analyzed, and inconsistencies are observed.

Halpern and Weissman [21] and Bruns and Huth [11] use logic and formal methods for access control policies. Halpern and Weissman [21] use a fragment of first-order logic called Lithium to describe and reason about policies. The authors describe two types of possible queries: (1) given a set of policies and an environment that provides all required information determine whether a particular action gets a response of permitted or forbidden and (2) given a set of policies determine whether these policies are consistent; i.e., no actions are both permitted and forbidden by the policies. It appears that there is no implementation of Lithium. By comparison, the implementation of the approach and reasoning capability is an important part of our work. Bruns and Huth [11] present a framework based on Belnap's four-valued logic in which an access request maps to one of the following four values: *grant*, *deny*, *conflict*, or *unspecified*. Bruns and Huth define an access control policy called PBel. They discuss policy composition using operators of Belnap logic. For instance, if one uses the "summing" of outcome in which an outcome of a policy is *grant*, and the outcome of another one is *gap* (e.g., no rule applies), then the combined policy creates the *grant* result. The concept of policy composition is akin to the combining algorithms in XACML. The authors provide examples but do not mention any experiments using their framework.

A few other papers use different formats to express or enhance XACML. Our enhancement includes the provision of state diagrams along with the pseudo-code format. Examples of this type of research follow. Li et al. [38] point out that XACML provides more flexible approaches of rule and policy combinations among policy languages, but even XACML cannot express formally several possible combinations. They provide a few possible approaches, such as weak-consensus, that XACML cannot express formally. Li et al. do not discuss the verification of AC properties of policies as we have discussed in this paper. Masi et al. [41] propose a formal XACML by using an alternative syntax in a BNF form and define formally the semantics of XACML. In contrast, our goal is to provide formal representations for rule- and policy-combining algorithms as the necessary step to verify properties of AC policies that use these combining algorithms. Bryans [12] discusses the representation of XACML policies using communicating sequential processes (CSP) and explains the use of this approach to verify whether a property, which is also expressed in CSP, holds in an XACML policy. The author uses a tool called failures-divergences refinement (FDR) that is designed to check models specified in CSP. Bryans uses a small grading example to show the approach. The author explains the representation of permit-overrides and deny-overrides in CSP and mentions that describing first-applicable represents a challenge in CSP because parallel operators in CSP are commutative.

Several research efforts are reported in the context of the semantic Web and its tools and languages. The scope of these efforts may or may not be related to XACML. Beimel and Peleg [9] also discuss the analysis of AC policies. The authors use the Web ontology language (OWL) and the semantic Web rule language (SWRL) as their specification languages and use an OWL-DL reasoner to provide analysis support. The scope of their work is not related to XACML and its combining algorithms. Kagal et al. [31] present Rein a policy framework to enable Web-based policy management. Rein has two parts: (1) a set of ontolgoies for describing policies and (2) a reasoning engine for RDF-S and OWL and for each supported rule language. Rein considers other policy languages, such as XACML, as domain-specific languages, and if the semantics of such languages can be described in RDF-S, OWL, or N3 rules, then they can be integrated with Rein. Therefore, the scope of this work is more related to

enabling a user to describe policies in different policy languages and reason over them by applying appropriate tools. Rei [32] is a policy language based on deontic concepts with the design goal of being grounded in a semantic Web language and with the capability of covering several domains. As a result, such a language must be capable of specifying a wide range of policies (including access control policies) and be adequate for expressing rights, prohibitions, obligations, dispensations (deferred obligations), and delegation. A policy engine for Rei has been developed using Prolog as a reasoning engine. One mechanism for resolving conflicts in Rei is *priorities* such that a rule or an entire policy has precedence over another rule or another policy. The *priorities* conflict resolution resembles to some extent the XACML combining algorithms, but the goal and scope of Rei is different from our work in this paper.

Several other efforts define various policy languages; two examples are mentioned next. Both efforts are concerned with the specification of authorization, whereas our work also has a component on the analysis of authorization. Woo and Lam [57] recommend the use of logic for the specification of authorization. The authors argue that separation of policies and mechanisms has been accepted because a mechanism deals with an actual implementation, whereas a policy specifies what needs to be done. The authors advocate the use of a logical framework, which is independent of an actual implementation, with formal semantics for policy specifications. Their proposed language is a many-sorted first-order logic, and the rule constructing in their proposal uses default logic. Becker et al. define an authorization language called SecPAL [7]. The authors indicate the inclusion of three features in their language: flexible delegations, domain-specific constraints (e.g., temporal constraint), and negation expressions.

The analysis of access control policies in general, but not in the context of XACML, is discussed by several authors. Jürjens et al. [30] describe UMLsec, which extends UML using stereotypes to enable security (both authorizations and authentications) specifications. They use a permission-based access control (i.e., associating permissions with entities) and annotate UML class diagrams with these permissions, which are subsequently translated into the language of a first-order theorem prover, such as SPASS, for analysis. Similarly, their work does not include rule- and policy-combining algorithms. Basin et al. [6] use OCL to write queries for policies of secureUML, which is based on RBAC. They introduce a tool called SecureMOVA, an extension of MOVA, to implement their approach. MOVA enables drawing UML class and object diagrams plus writing and evaluating OCL constraints. In comparison, our analysis is related to policies in XACML with the inclusion of combining algorithms and is also based on the specification of LTL properties.

Finally, we examine the new direction of the Margrave tool used by Fisler et al. [18] and described in Sect. 6.7. Nelson et al. [46] indicate that their work on Margrave extends "an earlier tool of the same name." The authors explain that their tool models policies in first-order logic, whereas many other firewall analysis tools use propositional models. Nelson et al. provide valuable analysis that is beyond the scope of our work in this paper. The authors explain that "Existing firewall analysis tools, including Margrave, largely ignore states," but the approach that we have shown takes advantage of states, and as a result we are able to express ordered versions of XACML combining algorithms.

8 Conclusion

This paper has first described a new formal representation for rule- and policy-combining algorithms (of policy languages) using state machines in which the transitions are governed by elements of the defined AC rules.

The approach can express and verify formally all-known policy- and rule-combining algorithms, a result not seen in the literature. The algorithms related to history of policy outcomes that include ordered-permit-overrides, ordered-deny-overrides, and only-one-applicable are expressed. The formal expressions of algorithms related to consensus that include weak-consensus, weak-majority, strong-consensus, strong-majority, and super-majority-permit are also described.

In addition, our approach supports automated formal verification of properties of AC policies that use these combining algorithms. We have used the access control policies and properties of a conference management system that has appeared frequently in the literature, and therefore, the comparison of results is possible. This case study is extended to include the verification of AC policies that use policy-combining algorithms involving history that have not been supported in the literature. We have also shown the verification results.

Acknowledgements We would like to thank K. Fisler and her co-authors [18] for making the XACML policies and the related properties of CONTINUE publicly available. We also thank the Natural Sciences and Engineering Research Council of Canada and the Ontario Research Fund for supporting this research.

Appendix 1: The CONTINUE policies

A prose description of twenty-five AC policies for CONTINUE conference management follows. The first-applicable combining rule within each of the following policies holds. These policies are available in the XACML format from the CONTINUE Web site.[4]

[4] http://www.cs.brown.edu/research/plt/software/margrave/versions/01-01/examples/continue/.

In CONTINUE, a *review-set* consists of four resources: *paper-review*, *paper-review-info*, *paper-review-info-reviewer*, and *paper-review-info-submissionStatus*. Similarly, a *review-content-set* consists of four resources: *paper-review-content*, *paper-review-content-rating*, *paper-review-content-comment sAll*, and *paper-review-content-commentsPc*. Furthermore, the first-applicable combining rule within each of the following policies holds.

Convention The dashes within names and the suffix *rc* (*rc* stands for *resource class*) are omitted; therefore, *paper-review* is used instead of *paper-review_rc*.

Policy one An *administrator* has permission to read and write *conference* resources, and a *pc chair* possesses permission to read these resources. A *pc member* at a meeting is permitted to read *conference* resources; an unidentified subject has no access to these resources.

Policy two An unidentified subject has access to read *conference info* resources. Any other permission to these resources is based on the same access rules applicable to *conference* resources.

Policy three A *pc member* has access to read *pc member* resources. An *administrator* possesses permission to write, create, and delete *pc member* resources. A *pc member* whose user-id is equal to the user-id of a *pc member* resources has no permission to perform any action on these resources. Any other individual's access to these resources follows the same rules for accessing *conference* resources.

Policy four A *pc chair* possesses permission to read and write *pc member assignment* resources, whereas a *pc member* is allowed to read his/her own assignments (i.e., a *pc member*'s user-id is equal to the user-id of a *pc member assignment* resources). An unidentified subject has no access to these resources. Other types of access to these resources follow the same rules for accessing *pc member* resources.

Policy five A *pc chair* has read and write access to *pc member conflict* resources, whereas a *pc member* is capable of reading his/her *conflict* resources. An unidentified subject has no access to these resources. In addition, other types of access to *pc member conflict* resources follow the same rules for accessing *pc member* resources.

Policy six Access to *pc member assignment count* resources is according to the rules for accessing *pc member* resources.

Policy seven A *pc chair* possesses permission to read and write *pc member info* resources, whereas a *pc member* has access to read and write his/her *pc member info* resources. An unidentified subject has no permission to access these resources. Furthermore, the same permission rules for accessing *pc member* resources hold for *pc member info* resources too.

Policy eight A *pc member* has write access to his/her *pc member info password* resources, and an administrator has the same permission whenever *pc member info password* resources are not pending. An unidentified subject does not possess any access to these resources. Additionally, the same permission rules for accessing *pc member info* resources also hold for accessing *pc member info password* resources.

Policy nine A *pc member* has access to read *pc member isChairFlag* resources, whereas a *pc member* whose user-id is equal to the user-id of these resources has no access to *pc member isChairFlag* resources. An unidentified subject has no access to these resources. Furthermore, the same permission rules for accessing *pc member info* resources also hold to access *pc member isChairFlag* resources.

Policy ten A *pc chair* possesses access to delete *paper* resources. A *pc member* has permission to read a paper if the *paper* is designated for a meeting; in addition, a *pc member* is allowed to create *paper* resources. Any other access to *paper* resources is based on the same rules for accessing *conference* resources.

Policy eleven A *pc chair* and a *pc member* are permitted to read *paper submission* resources, whereas a *sub-reviewer* is allowed to read only his/his own *paper submission* resources. In addition, the same permission rules for accessing *paper* resources are also applicable for accessing *paper submission* resources.

Note: A *pc member*, *P*, designates a *sub-reviewer*, *S*, to review *P*'s papers. *S* submits reviews for the assigned papers; after submitting these reviews, *S* has no future access to these reviews. *P* can access the reviews by *S* and modify and submit them. This arrangement makes *S* capable of using the conference management interface to read submitted papers and to write reviews. Otherwise, *P* has to make copies of submitted papers for *S* and retrieve *S*'s reviews without using the conference management interface.

Policy twelve Access to *paper submission info* resources follows the same criteria as those for accessing *paper submission* resources.

Policy thirteen The same rules for accessing *paper submission* resources are also applicable for accessing *paper submission file* resources.

Policy fourteen A *pc chair* in a meeting has read and write access to *paper decision* resources. Other criteria for accessing *paper decision* resources are based on the same rules as those for accessing *paper* resources.

Note In the following policies, the words "conflicted" and "unconflicted" indicate that people in a role may face conflicts of interest, such as when reading and writing reviews.

Policy fifteen A *pc chair* and an *administrator* are allowed to read and write *paper conflict* resources, whereas a *pc member* who is conflicted is permitted to read *paper conflict* resources. In addition, a *pc member* in a meeting has access to read *paper conflict* resources. An unidentified subject has no access to *paper conflict* resources. Furthermore, other types of access to *paper conflict* resources follow the same rules for accessing *paper* resources.

Policy sixteen A *pc chair* and an *administrator* are permitted to read and write *paper assignment* resources. An unidentified subject who is conflicted possesses no access to *paper assignment* resources. A *pc chair* in a meeting is allowed to read a *paper assignment* resource that is related to the meeting. An unidentified subject who is in the meeting is allowed to read *paper assignment* resources. An unidentified subject has no access to *paper assignment* resources. In addition, the same criteria for accessing *paper* resources are applicable for determining access permission for *paper assignment* resources.

Policy seventeen An unconflicted *pc chair* has all types of access to *paper-review* resources, whereas a *pc chair* in a meeting for particular *paper-review* resources is allowed to read only those resources. A *pc chair* is permitted to create and delete *paper-review* resources. A conflicted subject has no access to *paper-review* resources. An unconflicted *pc member* is permitted to read *paper-review* resources. All have all types of access to their own *paper-review* resources. All types of access are permitted to discussion phase *paper-review* resources. An unidentified subject who is assigned to *paper-review* resources and has already done his/her task is allowed to have any type of access to the resources, whereas an unidentified one assigned to particular *paper-review* resources has all types of access to them. An unidentified subject is not allowed to have any access to unassigned *paper-review* resources. Furthermore, other access rules to *paper* resources are also applicable to *paper-review* resources.

Policy eighteen A *pc chair* has all types of access to *paper-review info* resources; in addition, other types of access to *paper-review info* resources are based on the same criteria for accessing *paper-review* resources.

Policy nineteen A *pc member* is permitted to write, create, and delete *paper-review content* resources if a *pc member*'s user-id is equal to the user-id of the *paper-review content* resources, whereas a *sub-reviewer* is allowed to create *paper-review content* resources only if the *sub-reviewer* user-id is equal to the user-id of the *paper-review content* resources. Furthermore, other types of access to *paper-review content* resources follow the same criteria for accessing *paper-review* resources.

Policy twenty A *pc member* has permission to write *paper-review info submission status* resources if the *pc member*'s user-id equals the user-id of these resources and the content of these resources is already in place. Other types of access to *paper-review info submission status* resources are based on the same rules for accessing *paper-review info* resources.

Policy twenty-one All types of access to *paper-review-content-rating* resources are based on the same rules as those for accessing *paper-review content* resources.

Policy twenty-two All types of access to *paper-review content comments all* resources are based on the same rules as those for accessing *paper-review content* resources.

Policy twenty-three All types of access to *paper review content comments pc* resources are based on the same rules as those for accessing *paper-review content* resources.

Note: CONTINUE currently does not permit *comments* by *pc members* who have not written reviews for a paper, and therefore, have not read the paper in as much detail as the reviewers of that paper have but intend to provide *comments*, which are distinct from reviews, for authors.

Policy twenty-four All types of access to *paper-review-info-reviewer* resources are based on the same rules as those for accessing *paper-review info* resources.

Policy twenty-five A *pc chair* has read and write access to *is meeting flag* resources, whereas a *pc member* possesses only read access. In addition, other types of access criteria for *is Meeting flag* resources follow the same rules for accessing *conference* resources.

Appendix 2: Other policy-combining algorithm representation

Figure 19 shows the weak-consensus policy-combining algorithm according to the approach shown so far and is presented next.

Weak-majority [38] "A decision (permit or deny) wins if it has more votes than the opposite. Permit (deny, resp.) a request if the number of sub-policies permitting (denying, resp.) the request is greater than the number of sub-policies denying (permitting, resp.)."

Figure 20 shows the weak-majority policy-combining algorithm, and the corresponding state machine is presented in Fig. 21.

As mentioned previously, the algorithmic forms and state machines of the strong-consensus policy-combining algorithm, strong-majority policy-combining algorithm, and

```
initial state = state q00;
set PermitRes to false;
set DenyRes to false;
set ConflictRes to false;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-policy_i = false then
        move to state qi0 ;
    else
        move to state qi1;
        if Event_i Access(permit) = true for every element of EventResult then
            set PermitRes to true;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            set DenyRes to true;
            move to state denyRes;
        end
    end
end
if i = n and PermitRes = true and DenyRes = false then
    move to state permit;
else if i = n and DenyRes = true and PermitRes = false then
    move to state deny;
else if i = n and PermitRes = true and DenyRes = true then
    set ConflictRes to true;
    move to state conflict;
end
```

Fig. 19 The algorithmic form for weak-consensus policy-combining algorithm

```
initial state = state q00;
set NumPermit to zero;
set NumDeny to zero;
for i = 1 to n do
    // n = the number of policies in a policy set
    if premise-policy_i = false then
        move to state qi0 ;
    else
        move to state qi1;
        if Event_i Access(permit) = true for every element of EventResult then
            add one to NumPermit;
            move to state permitRes;
        else if Event_i Access(deny) = true for every element of EventResult then
            add one to NumDeny;
            move to state denyRes;
        end
    end
end
if i = n and NumPermit > NumDeny then
    move to state permit;
else if i = n and NumDeny > NumPermit then
    move to state deny;
end
```

Fig. 20 The algorithmic form for weak-majority policy-combining algorithm

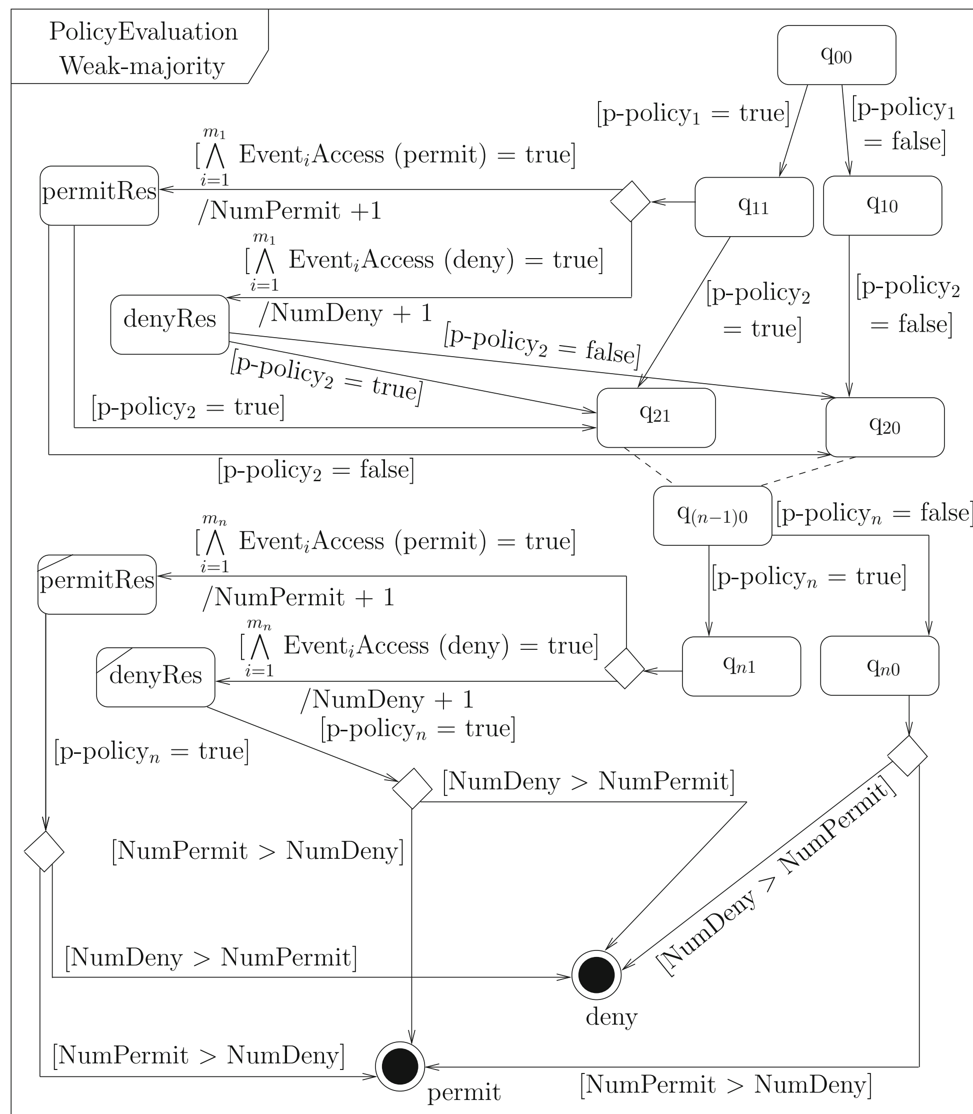

Fig. 21 A UML state machine for weak-majority policy-combining algorithm

super-majority-permit policy-combining algorithm are very similar to what has described so far.

Appendix 3: [34]: An AC rule format in Extended Backus–Naur Form (EBNF)

As explained in Sect. 3, one can view access control models as providing the basis for access control policies and rules. For instance, based on Fig. 2 and using the terms *resources*, *events*, and *agents* that we have described in Sect. 3, we can define an access control policy that consists of a single rule. The general form for this rule follows:

$$\text{ACRule} = (\text{AgentExp and ResourceExp and EventExp and AgeEveRel and ResEveRel}) \text{ implies EventResult}$$

The first expression defines an access control rule (ACRule) as an agent expression (*AgentExp*), a resource expression (*ResourceExp*), an event expression (*EventExp*), relationships related to agents and events (*AgeEveRel*), relationships related to resources and events (*ResEveRel*). The conjunctions of these expressions imply an event result (*EventResult*). We use Extended Backus–Naur Form (EBNF) to describe a general syntax of AC rules. Table 8 (ISO 14977) [54] shows the EBNF elements and their meanings.

Before describing the format of an AC rule in detail, an AC rule example is provided next. Terms enclosed by square quotation marks identify terminal elements in EBNF, and an EBNF rule termination is represented by a semicolon, as shown in Table 8. Therefore, we can rewrite the ACRule expression, just provided, as follows:

```
ACRule = (AgentExp "and" ResourceExp "and"
          EventExp "and" AgeEveRel "and"
          ResEveRel) "implies" EventResult;
```

The AC rule expressions are divided into six sections based on these six elements: *AgentExp*, *ResourceExp*, *EventExp*, *AgeEveRel*, *ResEveRel*, and *EventResult*. Each element is shown in detail in a following subsection with some prose explanations provided for a few expressions to make them easier to comprehend.

The first section, i.e., AgentExp, also shows the expressions for the general form of AC rules and presents some general expressions, such as class name (ClassName) and attribute name (attrName), that can be used by other sections.

AgentExp expressions

This section describes expressions that are mainly related to AgentExp, and some general definitions, such as attribute name (attrName), also apply to the other sections.

Figure 22 presents the expressions for this section, identified as "Part I."

- The expression starting with equA: *equA* can be expressions either about agents, or agent types, or agent groups (ATG), which are called agent-related expressions, or about the attributes of agents, or agent types, or agent groups (attrATG).
- The expression starting with equArep: zero or more agent-related expression(s) is (are) possible using '"and" and "or" connectives. An optional "not" can also precede *equArep*.
- The expression starting with attrATG: it identifies an attribute name, and its value in conjunction with an identification of *AgeAttIde*, which is defined shortly.
- The expression starting with AgeAttIde: an agent attribute identification can be either an agent designation, the class name of an agent, agent type, or agent group, or can be an agent designation along with the class name of the agent designation.

Table 8 Extended BNF (EBNF)

EBNF elements	Meaning
Unquoted words	Non-terminal symbol
"..."	Terminal symbol
(...)	Grouping
[...]	Optional symbols
{...}	Symbols repeated zero or more times
=	Defining symbol
|	Alternative
;	Rule termination

ResourceExp expressions

This section presents expressions for ResourceExp that are similar to AgentExp, which is just described, and therefore additional prose descriptions are not provided for this part.

The expressions for ResourceExp are shown in Fig. 23 and are identified as "Part II."

EventExp expressions

This section presents the EventExp expressions in Fig. 24. The expressions for EventExp, which are similar to the AgentExp and ResourceExp, are identified as "Part III."

AgeEveRel expressions

This section identifies the AgeEveRel expressions, as shown in Fig. 25, that are expressions about different agent and event relationships. The expressions for AgeEveRel are identified as "Part IV" in this figure. This figure can be explained as follows:

- The expression starting with AgeEveRel: the expression identifies the existence of one or more relationships involving agents and events. An optional negation is possible to indicate such a relationship is not true.
- The expression starting with AgeERel: the relationships involving agents and events can be between one of the following elements: agent type and event type, agent and event, agent type and agent group, agent type and agent, agent and agent group.
- The expression starting with AgeERelrep: this expression describes zero or more repetition(s) of relationships involving agents and events, as just described.

ResEveRel expressions

The ResEveRel expressions are presented in this section. These expressions describe different resource and event relationships and are similar to the AgeEveRel section. Figure 26 provides the expressions for ResEveRel that are identified as "Part V." These expressions are similar to AgeEveRel; therefore, the prose descriptions are not provided.

EventResult expressions

This section identifies and explains the EventResult expressions. Figure 27 shows the related EventResult expressions that are identified as "Part VI." This figure can be described as follows.

```
ACRule =  (AgentExp  "and"  ResourceExp  "and"  EventExp  "and"  AgeEveRel  "and"
                                ResEveRel)  "implies"  EventResult;
AgentExp =  ([uop]  equA  equArep);
uop =  "not";
equA =  ATG  |  attrATG;
equArep = {bop  [uop]  equA};
bop =  ("and"  |  "or");
ATG =  Agent "=" className  |  AgentTG "=" identifier
attrATG =  AgeAttIde"."attrNameValue  {"," AgeAttIde"."attrNameValue};
AgentDesignation =  Agent  |  AgentType  |  AgentGroup;
AgentTG =  AgentType  |  AgentGroup;
AgeAttIde =  ATG | ATClassName | AClassName | AGClassName | AgentDesignation
className =  "string";
identifier =  "instance"  "string";
instanceName =  "string";
Agent =  "var"  "string"  identifierA;
AgentType =  "var"  "string"  identifierAT;
AgentGroup =  "var"  "string"  identifierAG;
attrNameValue =  attrName  attrValue;
identiferA =  "Agent";
identiferAT =  "AgentType";
identiferAG =  "AgentGroup";
attrName =  "string";
attrValue =  relationN  valueNum  |  relationC  valueC;
relationN =  "<"  |  ">"  |  "≥"  |  "≤"  |  "="  |  "≠";
relationC =  "equals"  | "notEquals"
valueNum =  number;
number =  "integer"  |  "real";
valueC =  "character"  |  "string"  |  bool;
bool =  "True" | "False";
ATClassName =  identifier  "in"  (AgentType "=" identifier);
AClassName =  className  "in"  (Agent "=" className);
AGClassName =  identifier  "in"  (AgentGroup "=" identifier);
```

Fig. 22 AC rule definition in extended BNF, Part I

```
ResourceExp =  ([uop]  equR  equRrep);
equR =  RTG  |  attrRTG;
equRrep = {bop  [uop]  equR};
RTG =  Resource "=" className | ResourceTG "=" identifier ;
ResourceDesignation =  Resource  |  ResourceType  |  ResourceGroup;
ResourceTG =  ResourceType  |  ResourceGroup;
Resource =  "var"  "string"  idenitiferR;
ResourceType =  "var"  "string"  identifierRT;
ResourceGroup =  "var"  "string"  identifierRG;
attrRTG =  ResAttIde"."attrNameValue {"," ResAttIde"."attrNameValue};
identifierR =  "Resource";
identifierRT =  "ResourceType";
identifierRG =  "ResourceGroup";
ResAttIde =  RTG | RTClassName | RClassname | RGClassname | ResourceDesignation
RTClassName =  identifier  "in" (ResourceType "=" identifier);
RClassname =  className  "in" (Resource "=" className);
RGClassname =  identifier  "in" (ResourceGroup "=" identifier);
```

Fig. 23 AC rule definition in extended BNF, Part II

- The expression starting with EventResult: this expression describes the format of an event and its result. One or more events are possible.
- The expression starting with accETG: this expression describes the result of an event, or event type, or event group along with its class name.
- The expression starting with accETGrep: *accETGrep* describes zero or more repetition of accETG, which is just described.
- The expression starting with result: *result* can be either a *permit* or *deny*.

```
EventExpr =    ([uop]  ETG   ETGrep);
ETG =  Event "=" className | EventTG "="
                             identifier;
ETGrep = {bop   [uop]  ETG};
EventDesignation =  Event  | EventType | EventGroup;
EventTG =   EventType  |  EventGroup;
Event =   "var"  "string"  identifierE;
EventType = "var"  "string"  identifierET;
EventGroup = "var"  "string"  identifierEG;
identifierE =   "Event";
identifierET =   "EventType";
identifierEG =   "EventGroup";
```

Fig. 24 AC rule definition in extended BNF, Part III

```
AgeEveRel = ([uop]  AgeERel  AgeERelrep);
AgeERel = "RelATET"(ATClassName "," ETClassName)
          | "RelAE"(AClassName "," EClassName)  |
          "RelATG"(ATClassName "," AGClassName)
          | "RelAT"(AClassName "," ATClassName)
          | "RelAG"(AClassName "," AGClassName);
AgeERelrep = {bop   [uop]  AgeERel}
```

Fig. 25 AC rule definition in extended BNF, Part IV

```
ResEveRel = ([uop]  ResERel   ResERelrep);
ResERel = "RelRTET"(RTClassName "," ETClassName)
          | "RelRE"(RClassName "," EClassName)  |
          "RelRTG"(RTClassName "," RGClassName)  |
          "RelRT"(RClassName "," RTClassName)  |
          "RelRG"(RClassName "," RGClassName);
ResERelrep = {bop   [uop]  ResERel};
```

Fig. 26 AC rule definition in extended BNF, Part V

```
EventResult = ([uop]  accETG   accETGrep);
accETG = accessETG "=" result;
accETGrep = {bop   [uop]  accETG};
result = "permit"   |   "deny";
accessETG = (ETClassName | EClassName) "Access"
ETClassName = identifier   "in"   (EventType "="
                                  identifier);
EClassName = className   "in"   (Event "=" className);
```

Fig. 27 AC rule definition in extended BNF, Part VI

– The expression starting with accessETG: this expression defines the class name of an event type or event along with the word access.
– The expression starting with ETClassName: *ETClassName* distinguishes an identifier, previously defined, of an event type.
– The expression starting with EClassName: this expression describes the class name of an event.

References

1. Anderson, A.: A comparison of two privacy policy languages: EPAL and XACML. In: Proceedings of the 3rd ACM Workshop On Secure Web Services, pp. 53–60 (2006)
2. Arkoudas, K.: Athena, http://proofcentral.org/athena/ (2004)
3. Arkoudas, K., Chadha, R., Chiang, J.: Sophisticated access control via SMT and logical frameworks. ACM Trans. Inf. Syst. Secur. **16**(4), 17 (2014)
4. Ashley, P., Hada, S., Karjoth, G., Powers, C., Schunter, M.: Enterprise Privacy Authorization Language (EPAL 1.2). W3C Member Submission (2003)
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
6. Basin, D., Clavel, M., Doser, J., Egea, M.: Automated analysis of security-design models. Inf. Softw. Technol. **51**(5), 815–831 (2009)
7. Becker, M., Fournet, C., Gordon, A.: Design and semantics of a decentralized authorization language. In: Proceedings of the IEEE Computer Security Foundations Symposium (CSF), pp. 3–15 (2007)
8. Ben-Ari, M.: Principles of the Spin Model Checker. Springer, Berlin (2008)
9. Beimel, D., Peleg, M.: Using OWL and SWRL to represent and reason with situation-based access control policies. Data Knowl. Eng. **70**(6), 596–615 (2011)
10. Bray, H.: Payroll Website Still Not Secured. The Boston Globe, March 1 (2005)
11. Bruns, G., Huth, M.: Access control via belnap logic: intuitive, expressive, and analyzable policy composition. ACM Trans. Inf. Syst. Secur. **14**(1), 9 (2011)
12. Bryans, J.: Reasoning about XACML policies using CSP. In: Proceedings of the Workshop on Secure Web Services, pp. 28–35 (2005)
13. Constantin, L.: Twitter flaw gave third-party apps unauthorized access to private messages, researcher says. InfoWorld (2013)
14. Emerson, E.: The beginning of model checking: a personal perspective. In: Proceedings of the 25 Years of Model Checking, pp. 27–45 (2008)
15. Ferraiolo, D., Kuhn, D., Chandramouli, R.: Role-Based Access Control, 2nd edn. Artech House, London (2007)
16. Ferraiolo, D., Sandhu, R., Gavrila, S., Kuhn, D., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Trans. Inf. Syst. Secur. **4**(3), 224–274 (2001)
17. Ferraiolo, D., Kuhn, D.: Role-based access control. In: Proceedings of the National Computer Security Conference, pp. 554–563 (1992)
18. Fisler, K., Krishnamurthi, S., Meyerovich, L., Tschantz, M.: Verification and change-impact analysis of access-control policies. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 196–205 (2005)
19. Fisler, K., Krishnamurthi, S., Dougherty, D.: Embracing policy engineering. In: Proceedings of the Workshop on Future of Software Engineering Research (FoSER), pp. 109–110 (2010)
20. Geerts, G., McCarthy, W.: Policy-level specifications in REA enterprise information systems. J. Inf. Syst. **20**(2), 37–63 (2006)
21. Halpern, j, Weissman, V.: Using first-order logic to reason about policies. ACM Trans. Inf. Syst. Secur. **11**(4), 1–41 (2008)
22. Holzmann, G.: Parallelizing the Spin model checker. In: Proceedings of the International SPIN Workshop, pp. 155–171 (2012)
23. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
24. Holzmann, G.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
25. Holzmann, G., Joshi, R., Groce, A.: Swarm verification techniques. IEEE Trans. Softw. Eng. **37**(6), 845–857 (2011)

26. Hruby, P. with contributions by Kiehn, J., Scheller, C.: Model-Driven Design Using Business Patterns. Springer, Berlin (2006)
27. Hughes, G., Bultan, T.: Automated verification of access control policies using a SAT solver. STTT **10**(6), 503–520 (2008)
28. Jackson, D.: Software Abstractions: Logic, Language, and Analysis, Rev edn. MIT Press, Cambridge (2011)
29. Jha, S., Li, N., Tripunitara, M., Wang, Q., Winsborough, W.: Towards formal verification of role-based access control policies. IEEE Trans. Dependable Sec. Comput. **5**(4), 242–255 (2008)
30. Jürjens, J., Schreck, J., Yu, Y.: Automated analysis of permission-based security using UMLsec. In: Proceedings of the International Conference Fundamental Approaches to Software Engineering (FASE), pp. 292–295 (2008)
31. Kagal, L., Berners-Lee, T., Connolly, D., Weitzner, D.: Using semantic Web technologies for policy management on the Web. In: Proceedings of the Association for the Advancement of Artificial Intelligence (AAAI) Conference, pp. 1337–1344 (2006)
32. Kagal, L., Finin, T., Joshi, A.: A policy language for a pervasive computing environment. In: Proceedings of Policy, pp. 63–74 (2003)
33. Karimi, V.: A Uniform Formal Approach to Business and Access Control Models, Policies and Their Combinations. Ph.D. thesis, University of Waterloo (2012)
34. Karimi, V., Alencar, P., Cowan, D.: A uniform approach for access control and business models with explicit rule realization. Int. J. Inf. Secur. (2015). doi:10.1007/s10207-015-0275-z
35. Kern, A., Walhorn, C.: Rule support for role-based access control. In: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 130–138 (2005)
36. Kolovski, V., Hendler, J., Parsia, B.: Analyzing Web access control policies. In: Proceedings of the International Conference on World Wide Web (WWW), pp. 677–686 (2007)
37. Krishnamurthi, S.: The CONTINUE server (or, how I administered PADL 2002 and 2003). In: Proceedings of the International Symposium Practical Aspects of Declarative Languages (PADL), pp. 2–16 (2003)
38. Li, N., Wang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., Lin, D.: Access control policy combining: theory meets practice. In: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 135–144 (2009)
39. Mankai, M., Logrippo, L.: Access control policies: modeling and validation. In: Proceedings of the NOTERE Conference, pp. 85–91 (2005)
40. Martin, J., Odell, J.: Object-Oriented Methods: A Foundation, UML edn. Prentice Hall, Englewood Cliffs (1998)
41. Masi, M., Pugliese, R., Tiezzi, F.: Formalisation and implementation of the XACML access control mechanism. In: Proceedings of the International Symposium of Engineering Secure Software and Systems (ESSoS), pp. 60–74 (2012)
42. Motschnig-Pitrik, R., Kaasbøll, J.: Part-whole relationship categories and their application in object-oriented analysis. IEEE Trans. Knowl. Data Eng. **11**(5), 779–797 (1999)
43. Motschnig-Pitrik, R., Storey, V.: Modelling of set membership: the notion and the issues. Data Knowl. Eng. **16**(2), 147–185 (1995)
44. Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM **54**(9), 69–77 (2011)
45. Mouratidis, H., Giorgini, P., Manson, G.: When security meets software engineering: a case of modelling secure information systems. Inf. Syst. **30**(8), 609–629 (2005)
46. Nelson, T., Barratt, C., Dougherty, D., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis, In: Proceedings of the Large Installation System Administration Conference (LISA), pp. 1–18 (2010)
47. Ni, Q., Bertino, E.: xfACL: an extensible functional language for access control. In: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 61–72 (2011)
48. Ruys, T.: SPIN tutorial: How to become a SPIN doctor. In: Proceedings of the International SPIN Workshop, pp. 6–13 (2002)
49. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-based access control model. IEEE Comput. **29**(2), 38–47 (1996)
50. Schaad, A., Lotz, V., Sohr, K.: A model-checking approach to analysing organisational controls in a loan origination process. In: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 139–149 (2006)
51. Schaad, A., Moffett, J.: A lightweight approach to specification and analysis of role-based access control extensions. In: Proceedings of the ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 13–22 (2002)
52. Shanks, G., Tansley, E., Nuredini, J., Tobin, D.: Representing part–whole relations in conceptual modeling: an empirical evaluation. MIS Q. **32**(3), 553–573 (2008)
53. Toahchoodee, M., Ray, I: Validation of policy integration using Alloy. In: Proceedings of the International Conference on Distributed Computing and Internet Technology (ICDCIT), pp. 420–431 (2005)
54. The International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC): International Standard, ISO/IEC 14977. Information technology-Syntactic metalanguage-Extended BNF (1996)
55. Organization for the Advancement of Structured Information Standards (OASIS), Moses, T. (ed.): eXtensible Access Control Markup Language (XACML), Version 2.0 (2005)
56. Organization for the Advancement of Structured Information Standards (OASIS): eXtensible Access Control Markup Language (XACML), Rissanen, E. (ed.) Version 3.0 (2013)
57. Woo, T., Lam, S.: Authorizations in distributed systems: a new approach. J. Comput. Secur. **2**(2–3), 107–136 (1993)
58. Zeller, T.: Not Yet in Business School, and Already Flunking Ethics. The New York Times (2005)
59. Zhang, N., Ryan, M., Guelev, D.: Evaluating access control policies through model checking. In: Proceedings the International Conference on Information Security (ISC), pp. 446–460 (2005)