Multi-data-types Interval Decision Diagrams for XACML Evaluation Engine

Canh Ngo*, Marc X. Makkes*†, Yuri Demchenko*, Cees de Laat*
* University of Amsterdam
Science Park 904, 1089XH Amsterdam, The Netherlands
Email: {t.c.ngo, y.demchenko, delaat}@uva.nl
† TNO Information and Communication Technology
Eemsgolaan 3, 9727DW Groningen, The Netherlands
Email: marc.makkes@tno.nl

***Abstract*—XACML policy evaluation efficiency is an important factor influencing the overall system performance, especially when the number of policies grows. Some existing approaches on high performance XACML policy evaluation can support simple policies with equality comparisons and handle requests with well defined conditions. Such mechanisms do not provide the semantic correctness of combining algorithms in cases with indeterminate and not-applicable states. They ignore the critical attribute setting, a mandatory property in XACML, leading to potential missing attribute attacks. In this paper, we present a solution using data interval partition aggregation together with new decision diagram combinations, that not only optimizes the performance but also provides correctness and completeness of XACML 3.0 features, including complex logical expressions, correctness in indeterminate states processing, critical attribute setting, obligations and advices as well as complex comparison functions for multiple data types.**

Keywords*—*Access control, authorization, XACML, policy evaluation, decision diagram, multi-data-types Interval Decision Diagram (MIDD), interval partition processing, MIDD combination.

I. Introduction

Extensible Access Control Mark-up Language (XACML) is an authorization policy language in XML based on Attribute-Based Access Control model (ABAC). It composes policies from set of attribute criteria joined by logical operators to decide if authorization requests are granted. XACML is scalable in arrange policies in hierarchy order that can be combined and extended vertically by conflict resolution algorithms. In addition, XACML supports delegations, obligations and advices, that makes it applicable in many areas such as networking, grids, clouds, enterprise organization and management. However, the growth of policies to address system scales can increase complexity of the policy repository which decreases policies evaluation performance. Motivated from the related work on decision diagrams [1]–[3], policy evaluations [4], [5] and the high performance requirements in XACML authorization services for on-demand complex resource provisioning [6], cloud-based and virtualized infrastructure services provisioning [7], we present a high performance XACML policy evaluation solution using multi-data-types interval decision diagrams (MIDD) that provides policy evaluation correctness and completeness. Our contributions are summarized as follows:

- Analyze XACML target expressions as complete logical expressions, modeling combining algorithms as deterministic finite automaton (DFA) which facilitate the XACML policy evaluation approach.
- Define interval processing and MIDD operations to efficiently and precisely evaluate the complex logical expressions.
- Support complex comparison functions for continuous data types.
- Introduce MIDD and X-MIDD, the new decision diagram data structures for XACML evaluation. They can preserve the correctness of combining algorithms semantic in handling indeterminate and not-applicable states in standards [8], [9].
- Support critical attribute setting specified in XACML standards which is missing in prior work [4], [5].

The rest of the paper is organized as follows. After the related work in section II, we identify the drawbacks in prior work in section III. Section IV analyzes XACML evaluation algorithms that provides the basis for the proposed solution. Section V formulates the approach to evaluate the complete logical expressions using interval decision diagrams. In section VI, we define fundamental operations to process intervals, partitions and decision diagrams. These materials are used in our solution to optimize XACML evaluation in section VII. The solution then is analyzed and validated in section VIII. Finally section IX contains conclusions and discusses our future work.

II. Related Work

Prior work on XACML focuses on verification, semantic correctness and policies management assistance to detect and remove redundancy, optimize policy tree structure to improve evaluation performance indirectly [10]–[12]. Others provide new features such as policy combining language [13], policy decomposition [12], or policy integration [14].

Recent work solves the policy evaluation problem directly by different ways [4], [5], [15]. Marouf et al. [15] reorder most frequent applicable policies and rules in the policy tree using clustering technique based on statistics from past requests and matched policies. The drawback with this work is the assumption of access requests following a uniform distribution. The recent approaches [4], [5] use decision diagrams techniques to

get better performance than the standard SunXACML engine [16]. However, they only solve the part of the policy evaluation problem due to the lack of the correctness in error decisions handling, complex logical expressions as well as unsupported XACML functionalities like policy obligations and critical attribute setting [8], [9]. In subsequent sections, we present a new solution with X-MIDD data structure to solve such drawbacks.

III. Problem Statement

Decision diagram is the popular tool in model checking, software verification and hardware design [1], [2], as well as policy verification [11] and firewall packet filtering [17]. Generally, it is an acyclic, direct graph $G(V, E)$ where set of nodes V represents input variables and set of edges E represents predicates of variables at internal nodes. Using the graph, a decision is made through the path from the root node to a leaf node.

XEngine [4] and graph-based approach [5] are the most recent works to use decision diagrams in XACML policy evaluation. Their key idea is to transform the target expressions scattered in the policy tree into decision diagrams as an index mechanism for applicable rules. Each variable is equivalent to an attribute and edge conditions are extracted predicates from target expressions. Liu et al. [4] numericalize all presented values in policies to integers, while the equality comparison of multiple data types is transformed to the one of integer data type. Although this technique significantly improve evaluation performance, it assumes all attribute predicates in the policies must be equal operators. Second, due to numericalization process, all applicable attribute values in requests must be present in defined policies. These limitations make xEngine applicable only for a specific class of XACML policies. It also ignores obligations processing as well as incorrectly handled indeterminate values. However, the performance is substantially improved when numerical comparisons are much faster than different data types comparisons.

Ros et al. [5] extend the work with interval techniques. They keep original data types of attributes with more supported comparison operators than xEngine. The approach uses two trees: the Matching Tree (MT) is a decision diagram built up from extracted predicates in target expressions, and the Combining Tree (CT) data structure at MT's leaf nodes. A CT is the subset of the XACML policy tree containing only applicable elements (policy-set, policy or rule) with their *"Effect"* values. The CT evaluation follows the defined XACML rule/policy combining algorithms. It is different from [4] when xEngine stores flat of applicable rules rather than the subset of policy tree.

However, the approach in [5] still has following drawbacks:

A. Correctness problem

XACML policy decisions may not only have binary values Permit or Deny, but also contain other intermediate values presented in Table I. The XACML 3.0 extends with different indeterminate values to better handle intermediate states. Policies uses combining algorithms to aggregate their children decisions, so an incorrect child decision could affect the final result of the whole policy tree. The evaluation process becomes complicated to effectively handle such situations.

1) Missing attributes processing: In request contexts, XACML use *MustBePresent* property in the *AttributeDesignator* and *AttributeSelector* elements to mark if the attribute is critical or optional. If an optional attribute is missing from a request, the retrieval engine returns an empty bag of values, then expression evaluation is NA. In case with the critical attribute, an exception is raised with an IN decision. This feature is to prevent missing attribute attacks, the one to circumvent authorization engine by crafting special requests with some missing attributes. For example a policy uses *DenyOverrides* algorithm (DO)[1] and contains two children rules r_1 and r_2: r_1 defines requests with *role* attribute be *"guests"* will be denied, and r_2 returns permit for suitable attribute conditions. Without critical attribute setting, a crafted request missing *role* attribute can deceive r_1 from IN_D to NA result, and the final policy decision could be $DO(NA, P) \rightarrow P$, rather than $DO(IN_D, P) \rightarrow IN_{DP}$.

The graph-based approach [5] does not handle this feature. When finding the matching path in the MT, if any attribute is missing from the request context, the evaluation is forked and may reach multiple terminating nodes. This is the incorrect processing because terminating nodes contain applicable rules, but with missing attributes, they are not applicable. Although the paper mentions about CTs merging, its detail operation is not clearly defined. So, we argue that this approach has a drawback in missing attribute processing.

2) Indeterminate states handling: XACML use indeterminate states to handle different error types in target and condition expression evaluations, including fails in attribute retrievals due to either network connections or syntax errors, missing attributes in requests. In any cases, returned IN values must be properly handled following combining algorithms in [8], [9]. For example with policy in Fig.1, assuming price attribute in R01 rule is marked as *MustBePresent*, if a request contains wrong data-type for this attribute value, e.g. (1085BL, 9am, 'any'), decisions for R01 and R02 will be IN_P and D, respectively. The P0 policy decision will be $PermitOverrides(IN_P, D) \rightarrow IN_{DP}$.

However, Ros et al. [5] only count for applicable rules and ignore rules with IN values. In the above example, only R02 rule is applicable, then their evaluation will be the wrong decision D value. So the combining tree approach [5] cannot handle all requests with indeterminate states handling as in the standards.

B. Data interval processing

The detail solution in [5] contains set of interval processing operations such as intersect intervals and subtract intervals. However, it incorrectly defines these operations to work in every situation. For example with subtract intervals (Algorithm 3, line 17 [5]), it assumes that the subtraction result cam only be an interval, such as $(3, 6) \setminus (1, 4) \rightarrow (4, 6)$. But with the case $(-\infty, 5) \setminus (1, 2)$, it should return two disjoint intervals $\{(-\infty, 1), (2, 5)\}$. Thus the interval path IP, defined as a list of intervals containing exactly one interval for each different attribute id, as well as related algorithms should be fixed.

[1] Section IV and appendix A give more detail on XACML combining algorithms.

Fig. 1. A Sample XACML 3.0 Policy Tree

Otherwise the result MT does not have the same matching semantic with the original target expressions.

Based on analysis in section IV, our solution can process complex logical expressions properly with our defined interval partition operators in section VI. We then use the modeled DFAs for combining algorithms to create the X-MIDD structure to solve above correctness issues.

IV. XACML Analysis

XACML policies are organized as a hierarchy, containing policy-sets, policies and rules. Each one has a target expression as the criteria for requests. The returned decision is either specified in its *"Effect"* property if this is a rule, or by combining its children's decisions if this is a policy-set or a policy. A sample policy tree is in Fig.1.

A. Target Expression Processing

A target expression essentially is a compound logical expression of attributes as follows:

$$T(X) = \bigwedge_i \left[\bigvee_j \left(\bigwedge_k m_k \right) \right] \quad (1)$$

in which m_k is a match expression, denoted by a tuple of three elements: (x, f, v) where x is the attribute id, v is an attribute value and f is a two-operands predicate Boolean function. The $T(X)$ is said *"matched"* when (1) returns *"true"*.

In Fig.1, with the *postal-code* attribute is denoted by x_1, *time* by x_2 and *price* by x_3, the target expression of rule "R02" is:

$$[(x_1 = "1080AB") \wedge (x_2 = 17)] \vee [(x_1 \geq "1085BL") \wedge (x_1 \leq "1095CJ") \wedge (x_2 = 9)] \vee [(x_1 = "1098XH") \wedge (x_2 = 12)]$$

TABLE I. XACML Decision States

States	Abbreviation
Permit	P
Deny	D
Indeterminate	IN
Indeterminate{P}	IN_P
Indeterminate{D}	IN_D
Indeterminate{DP}	IN_{DP}
NotApplicable	NA

$X = \{x_1, x_2.., x_n\}$ is a vector of attribute values denoting for a XACML request. For single-valued requests, each attribute-id has only a value, while in multi-valued requests, an attribute may contain list of values.

Policy evaluation is the process to find applicable rules then combine their decisions. A rule r in the policy tree is applicable for a given request X only if all its ancestors in the path from the policy root P_0 to policy P_k are also applicable:

$$\bigwedge_{i \in \{P_0..P_k, r\}} T_i(X) = true \quad (2)$$

Evaluation engines often process each target expression $T_i(X)$ consequently. Our approach is to evaluate (2) efficiently to gain high performance while still preserving XACML correctness features. The general method is to collect and reduce all $T(X)$ expressions distributed in the policy tree. Data interval techniques used in [5] and our paper extract attributes' intervals from such logical expressions (e.g. $(x > 3) \wedge (x < 5)$, can extract the interval $(3, 5)$).

However, the [5] only supports simple target expressions. It uses predicate (P), predicate path (PP) and predicate path list (PPL) concepts that can only aggregate expressions with form: $\bigvee_j (\bigwedge_k m_k)$ (Algorithm 1 in [5]). We solve the problem completely by defining intervals, partitions and their equivalent operators in section VI on generic data-types. They are used to create decision diagrams, which can construct and aggregate logical expressions (2).

B. Combining Algorithms in XACML

XACML defines set of algorithms to combine a policy children's decisions. Due to their sophisticated processing with multiple inputs (Table I), it is unfeasible to describe by binary operators. In this paper, we choose to represent them by using DFAs. This approach is flexible because it's able to describe current algorithms in [8], [9] or any other combining algorithms.

The DFA representing a combining algorithm is defined as follows:

- States: $Q = \{P, D, IN_P, IN_D, IN_{PD}, NA\}$
- Input symbols: $\Sigma = \{P, D, IN_P, IN_D, IN_{PD}, NA\}$
- Start state: NA
- Accept states: Q
- Transition function: $\delta : Q \times \Sigma \rightarrow Q$, which can be defined by a transition table.

Appendix A demonstrates some transition tables for combining algorithms in XACML 3.0 [9]. While most combining

algorithms are symmetric, the "First-applicable" algorithm is asymmetric.

In next sections, we symbolize (2) as decision diagrams and use above DFAs to construct a policy decision diagram from its children.

V. Multi-data-type Inteval Decision Diagrams

There are several approaches to evaluate (1), (2) efficiently. Prior work on model checking and verification tools uses binary decision diagram (BDD) [1] and interval decision diagrams (IDD) [2] but they cannot adapt our requirements. They only support binary or discrete domain values (integer data type in [2]), but XACML and other attribute-based policy languages are mostly continuous data types (e.g. string, double, date, time) or equally comparable data type (binary, URI) while work in [4] only supports the later. In this section, we formulate the MIDD mechanism to support such above data types.

It's possible to use BDD techniques when seeing m_k as binary variables. However, because of composing from combination of attributes, the number of m_k expressions is much higher than the number of attributes, leading the time and space complexity in this case are higher than in the MIDD approach.

A. Concepts

Denoting a multi-variable logical function with following signature:

$$f : D_1 \times D_2 \ldots \times D_n \rightarrow \{true, false\} \tag{3}$$

where D_i is a total order domain representing a continuous data-type in XACML. Denoting vector $X = (x_i | i = 1..n, x_i \in D_i)$, expression (3) is also seen as $f(X) \rightarrow \{true, false\}$

Definition 1. *A data interval $I \subset D_i$ is a range of values in the domain D_i which is formed by two endpoints.*

Definition 2. *An interval partition P is a set of disjoint intervals in the domain D_i: $P = \{I, I \subset D_i : \forall I_i, I_j \in P, i \neq j, I_i \cap I_j = \emptyset\}$*

Given an interval partition P, the denotation $x_i \in P$ means that $\exists I \in P$, s.t $x_i \in I$.

We define a boolean function $h_{x_i}(P)$ as:

$$h_{x_i}(P) = \begin{cases} 0 & \text{if } x_i \notin P \\ 1 & \text{if } x_i \in P \end{cases}$$

Function f in (3) is called independent with $x_i \in X$ in the interval partition P when:

$$\forall a, b \in P, f(X|_{x_i:=a}) = f(X|_{x_i:=b}) \tag{4}$$

with a, b are variable values. In this case, we denote $f_{x_i^P}$ as the partial function:

$$f_{x_i^P} := f(x_1 ..., x_{i-1}, b, x_{i+1}, .., x_n)|_{\forall b \in P} \tag{5}$$

Given a domain D_i, the set of partitions $P(D_i) = \{P_1, P_2 ..., P_{d_i}\}$ is called to cover the domain D_i when

$$D_i = \bigcup_{P \in P(D_i)} \left(\bigcup_{I \in P} I \right) \tag{6}$$

The cover $P(D_i)$ is disjoint if there's no common interval between them:

$$\forall i, j \in [1, d_i], i \neq j : P_i \cap P_j = \emptyset \tag{7}$$

According to Boole-Shannon expansion, a function f can be decomposed to set of partial functions in respect of variable x_i against a disjoint, covered partition $P(D_i)$

$$f(X) = \bigvee_{P \in P(D_i)} h_{x_i}(P) \wedge f_{x_i^P} \tag{8}$$

Each partial function $f_{x_i^P}$, can also be decomposed in respect to other variables, until all of them are free. We can symbolize f as a decision diagram $G(V, E)$ with following properties:

- G is a rooted, directed acyclic graph with node set V having two types of nodes: internal nodes and leaf nodes containing Boolean values.
- The internal node $v_{x_i} \in V$ has a variable $x_i \in D_i$ of the function f. Each out-going edge $e_{x_i} \in E$ is a predicate of x_i containing a partition $P_{x_i} \in P(D_i)$. The predicate is *true* when a value $v_{x_i} \in P_{x_i}$.
- Sub-graph of the node v_{x_i} is a partial function described in (8).

We call the graph $G(V, E)$ with n internal node levels above as the MIDD.

B. From Logic Expression to MIDD

We can reduce and transform expression in (2) to a MIDD as follows:

1) For each m_k, extract a data interval I_k for the attribute x_i.
2) For the expression $\bigwedge_k m_k$, we aggregate intervals of the same attribute using the intersect operator. The result is a list of interval partitions $\{P_1, P_2 .., P_n\}$, exactly one partition P_i for an attribute x_i. Because of independent with all attributes, the expression can be symbolized as a MIDD according to (8).
3) With defined conjunctive and disjunctive operators over MIDDs, we can construct a MIDD representing the expression in (2).

VI. Interval Processing and MIDD Combination

This section defines interval processing operations and MIDD composition operators as follows:

A. Interval Partition Processing

Definition 3. *A reduced interval partition has least number of intervals comparing to other interval partitions having the same data ranges.*

Given an interval partition, we find and combine intervals that have adjacent ranges repeatedly until no adjacent-range interval is found. The result is a reduced interval partition.

IDD operations [2] are only support for integer data type. So we define following operations for continuous data types on two reduced interval partitions P_1 and P_2:

1) Union: $P = P_1 \curlyvee P_2$ has below properties:

- P is a reduced interval partition.
- All values belong to either partitions P_1 or P_2 also belong to P: $\forall v \in P_1 \cup P_2, v \in P$

For example, with $P_1 = \{[-3, 4.5], [6.3, 8]\}$, $P_2 = \{(2, 5.1], (7.5, 9]\}$, we have $P_1 \curlyvee P_2 = \{[-3, 5.1], [6.3, 9]\}$

2) Intersect: $P = P_1 \curlywedge P_2$ has following properties:

- P is a reduced interval partition.
- P is composed from all common values of P_1 and P_2: $\forall v \in P_1 \cap P_2, v \in P$

With P_1 and P_2 in the above example: $P_1 \curlywedge P_2 = \{(2, 4.5], (7.5, 8]\}$

3) Complement: $P = P_1 \ominus P_2$ is an interval partition that:

- P is a reduced interval partition.
- It contains values of P_1 but not P_2: $\forall v \in P_1 \setminus P_2, v \in P$

For P_1 and P_2 in the above example: $P_1 \ominus P_2 = \{[-3, 2], [6.3, 7.5]\}$

B. MIDD Composing Algorithms

We define MIDDs algorithms equivalent to conjunctive and disjunctive logical operators. With these algorithms, we can compose a MIDD representing a complex logical expression in (2).

We define a node N in MIDD as the data structure of $n := (x, C)$ in which x is the variable of the node. The C is the set of tuple (c, e) representing an edge e connecting N to a descendant node c. Edge e contains a reduced interval partition $e.P$ as the predicate of the variable x. The leaf nodes contain *"true"* value with the implicit evaluation that if it cannot find any out-going edge at an internal node, the evaluation result will be *"false"*.

Variable ordering can affect the complexity of the MIDD, that we leave it for future work. Currently we choose an order in which variables appear in the policies.

Given two logical expressions f_1 and f_2 having equivalent MIDDs M_1 and M_2, combining operators on MIDDs are defined as follows:

Definition 4. *Conjunctive join MIDDs: The $M = M_1 \wedge M_2$ is a MIDD representing the logical expression $f = f_1 \wedge f_2$.*

Definition 5. *Disjunctive join MIDDs: The $M = M_1 \vee M_2$ is a MIDD representing the logical expression $f = f_1 \vee f_2$.*

The pseudo-code for these operators are shown in Algorithms 5 and 6, respectively.

Defined operations on intervals and MIDD can be used to parse not only XACML but also other attribute-based policy languages using logical expressions.

VII. SNE-XACML

A. Scope and Design Goals

We implement MIDD processing in our SNE-XACML evaluation engine. This section contains mechanisms to create a MIDD from the XACML logical expression (2) and to transform the original MIDD into an extension structure presenting the XACML 3.0 policy semantic, known as the X-MIDD. Beside having much higher evaluation performance, the X-MIDD could handle missing features from prior work as follows:

- Support complex comparison functions for continuous data types.
- Handle complete logical expressions defined in policies.
- Preserving original combining algorithms semantic in handling IN and NA states in XACML 3.0. This mechanism is also compatible for version 2.0.
- Supporting critical attribute feature: we assure this feature is compliant with XACML standards.

Besides that, our solution could support other XACML features such as obligations and advices handling, multi-valued attributes policies.

The steps to create X-MIDD are as follows:

- Extract and build MIDDs representing applicable rules as in (2).
- Transform the MIDD into the X-MIDDs of applicable rules along with internal nodes states, XACML decision values, conditions and obligations at leaf nodes.
- Combine rules' X-MIDDs using combining algorithms DFAs in section IV to construct the final X-MIDD instance representing the root policy.

The final X-MIDD instance is used for policy evaluation against incoming requests similar to MIDD.

B. Creating MIDD from a Target Expression

In XACML 3.0, the AllOf element has a list of match expressions, that represents the logical formula $\bigwedge_k m_k$. The AnyOf element consisting of a list of AllOf elements defines the $\bigvee_j (\bigwedge_k m_k)$ formula and finally, the target expression is described as in (1).

The Algorithm 1 creates the MIDD from a AllOf element. As described in section IV, a match expression m is a tuple of (x, f, v). The variable x contains a critical property s representing the *"MustBePresent"* boolean value.

In the algorithm, at first we extract and aggregate intervals belonging to each variable x using the function $I' := restrict(I, f, v)$. This function restricts the interval I for given function f and value v: $I' = \{x | x \in I; f(v, x) \rightarrow true\}$. For example, with $I = (-5, 8), restrict(I, \geq, 3) \rightarrow [3, 8)$.

From list of intervals IL having exactly one interval per variable, we build a MIDD that has only a path from the root to a true-leaf-node. Each node N in the path has a variable

```
Input: List of Match expressions: M = {m_1, m_2.., m_k}
Output: Result MIDD
1  begin
2      IL ← ∅;
3      foreach (m_i ∈ M) do
4          I ← IL[m_i.x];
5          I ← (I = null) ? (−∞, +∞) : I;
6          IL[m_i.x] ← restrict(I, m_i.f, m_i.v);
7      end
8      root ← null; N ← null;
9      foreach (attribute x ∈ IL.indices) do
10         s ← x.critical;
11         N' ← (x, s, {(null, IL[x])});
12         if (root = null) then
13             root ← N; N ← root;
14         end
15         else
16             N.addChild(N'); N ← N';
17         end
18     end
19     N.addChild(true-leaf-node);
20     return root;
21 end
```

Algorithm 1: The *parseAllOf* function

x, an out-going edge having an interval $\{IL[x]\}$ and a critical state s, which then is used for indeterminate processing.

The Algorithm 2 creates the MIDD from a target expression in (1) by using conjunctive and disjunctive join MIDD operators.

```
Input: List of AnyOf expressions: A = {a_1, a_2, ..a_k}
Output: A MIDD D
1  begin
2      D ← null;
3      foreach (AnyOf element a ∈ A) do
4          d ← null;
5          foreach (AllOf element ae ∈ a_i) do
6              d' ← parseAllOf(ae);
7              d ← disjunctive_join(d, d');
8          end
9          D ← conjunctive_join(D, d);
10     end
11     return D;
12 end
```

Algorithm 2: The *parseTarget* function

C. Building X-MIDD for Applicable Rule

In Algorithm 3, given a rule r, we construct the MIDD representing the expression as in (2). Then we transform it to the X-MIDD containing r's XACML decision using *transform* function. This function performs as follows:

- Replace the true-leaf-node with the decision-leaf-node containing rule's effect value $n.E$, condition expression $n.C$, applicable obligations $n.O$ and advices $n.A$.
- Critical attribute priority processing: Given a node $N := (x, s, C)$, if the variable x is critical, set default returned decision $N.s$ is either IN_P or IN_D, depending on r's *"Effect"* value. Otherwise, $N.s$ receives NA value by default.

If the processing node is either a policy or a policy-set, we parse its children recursively to obtain X-MIDDs and combine them using *combine* algorithm.

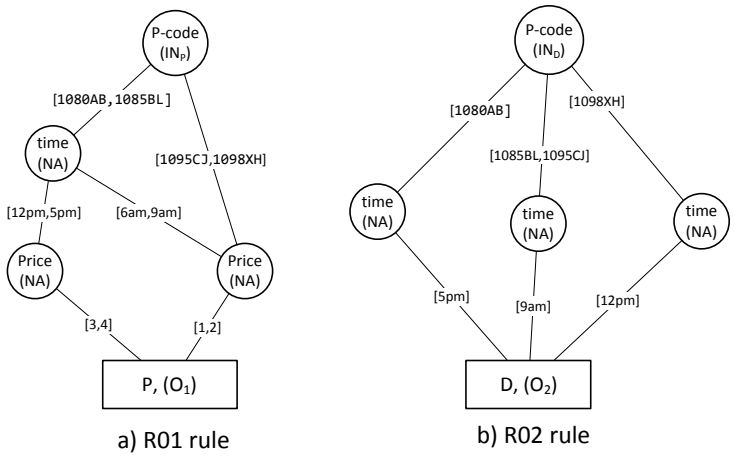

Fig. 2. X-MIDDs for rules

For sample policy tree in Fig.1, two rules "R01" and "R02" have their equivalent X-MIDDs shown in Fig.2. Because the attribute *postal-code* is marked as critical, the $N.s$ values of this variable are IN_P for "R01" rule and IN_D for "R02" rule.

```
Input: A policy node in the policy tree: n
       MIDD from ancestor's node: D_a
Output: X-MIDD represening policy node n: XD
begin
    D ← parseTarget(n.T);
    D ← conjunctive_join(D, D_a);
    if (n is a rule) then
        XD ← transform(D, n.E, n.C, n.O, n.A);
    end
    else
        foreach (n_i ∈ n.children) do
            d ← parse_policy_node(n_i, D);
            XD ← combine(XD, d, n.CA, n.O, n.A);
        end
    end
    return XD;
end
```

Algorithm 3: The *parse_policy_node* function

D. Combining X-MIDDs

The *combine* algorithm 4 combines X-MIDDs of policy/policy-set children using DFAs in section IV.

We denote $M.P$ as the union of all partitions of node M's out-going edges: $M.P \leftarrow \curlyvee\{\forall e.P, (c, e) \in M.C\}$

In the algorithm, if both M_1 and M_2 are leaf nodes having decisions, they are combined using DFA with matching obligations and advices.

Otherwise, if roots of M_1 and M_2 have the same variable, the root of new MIDD also contains this variable. The default returned decisions from M_1 and M_2 are combined using DFA. Its children is the combination of each of M_1 and M_2 children, respectively, aligned with each interval in the union interval partition $P = M_1.P \curlyvee M_2.P$.

If a M_i (say L) has lower variable order than the other (say H), we combine H with each child of L and add the output as the child of result MIDD M. We also add a new edge as the complement of all union-ed L's partitions to connect to H.

Input: X-MIDDs M_1, M_2; combining algorithm CA; obligations O, advices A
Output: Combined X-MIDD

begin
  if *(all M_i are leaf nodes)* **then**
    $d \leftarrow DFA_{CA}(M_1.d, M_2.d)$;
    // combine all obligation, advices and conditions **return** $M_{leaf} := (d, O, A, C)$;
  end
  else
    if *($M_1.var \equiv M_2.var$)* **then**
      $M \leftarrow (M_1.var, DFA_{CA}(M_1.s, M_2.s), \emptyset)$;
      $P = M_1.P \curlyvee M_2.P$;
      foreach *(interval $I \in P$)* **do**
        $c_1 \leftarrow M_1.C[I]; c_2 \leftarrow M_2.C[I]$;
        $c \leftarrow combine(c_1, c_2, CA, O, A)$;
        $e.P \leftarrow \{I\}$;
        $M.C \leftarrow M.C \cup \{(c, e)\}$;
      end
    end
    else
      $L \leftarrow M_i$ that has lower variable order;
      $H \leftarrow M_j$ that has higher variable order;
      $M \leftarrow (L.var, L.s, \emptyset)$;
      foreach *($(c, e) \in L.C$)* **do**
        $c' \leftarrow combine(c, H, CA, O, A)$;
        $M.C \leftarrow M.C \cup \{c', e\}$;
      end
      $\overline{P} \leftarrow \{(-\infty, +\infty)\} \ominus L.P$;
      $e'.P \leftarrow \overline{P}$;
      $M.C \leftarrow M.C \cup \{H, e'\}$;
    end
  end
  return M;
end

Algorithm 4: The *combine* function

Using the *parse_policy_node* algorithm for the root of policy tree, we can create a X-MIDD using for the evaluation process. The policy in Fig.1 has its X-MIDD shown in Fig.3.

E. X-MIDD Evaluation and Multi-valued Attributes

A single-valued request X is defined as a list of attribute values, exactly one value per attribute, $X = \{x_1, x_2..., x_n\}$. Evaluating X against policies is to find the matching path in the equivalent X-MIDD. At a node, finding the matching edge can be done either using sequential search or binary search, since out-going edges of an internal nodes having ordered interval partitions. We currently support single-valued requests.

If the matching path found, the engine reaches the leaf node. Here, if the condition C is *"true"*, the result is the decision value and equivalent objects (i.e. obligations, advices). Otherwise, it returns NA. If the matching path is not found due to missing attributes or no applicable edge at the node N, the evaluation return $N.s$, the default returned-decision, either in $\{NA, IN_D, IN_P, IN_{DP}\}$.

XACML allows multi-valued attribute requests, where an attribute can store a list of values (e.g. a person can have several roles: employees, managers, etc). However, current XACML behavior on multi-valued requests processing has some concerns. With policy in Fig. 1, given a request $X = \{"1085BL", \{10, 19\}, 4\}$ with the *time* attribute can either *10am* (10) or *7pm* (19), the evaluation following XACML standards [8], [9] such as SunXACML [16] claims that *R01* is the applicable rule. It has such result because each value in the bag of *time* attribute $\{10, 19\}$ is checked with each match expressions: the value 10 passes the condition $time < 17$, and value 19 passed the condition $time > 12$. While in practical, we expect *R01* is not applicable when it requires a *time* value passes both conditions, not two separate values.

Prior work in [4], [5] decomposed multi-valued requests in set of single-valued requests to evaluate and combine all of applicable rules. As illustrated in the above example, this processing does not follow XACML standards.

For multi-valued policies supports in [4], [5], we argue that they only used in XACML condition expressions, not in target expressions due to the value v in $m_k := (x, f, v)$ can only receive a literal value [8], [9]. Because prior approaches [4], [5] only deal with target expressions, their supports are redundant. Our solution evaluates condition expressions at leaf nodes, so it's possible to handle multi-valued attributes in policies.

VIII. Evaluation

A. Features comparison

As analysis in previous sections, our solution supports following features missing from prior work [4], [5]:

- Handling complete logical expressions in (1) using in attribute-based policies.
- Preserving original combining algorithms semantic in handling indeterminate and not-applicable states: prior work handles well on simple *Permit* or *Deny*, but incorrectly for other states.
- Critical attribute setting: we are the first work to support this feature with high performance evaluation.

B. Complexity

1) Space complexity: Suppose a set of policies using n attributes $\{a_1, a_2.., a_n\}$, $a_i \in P_i$. Assuming that the domain P_i has k_i different values appearing in the policies, so P_i can be separated into at most $2k_i + 1$ intervals or partitions, including open and degenerate intervals. So the X-MIDD has at most $2k_i + 1$ out-going edges from any node at level l_i.

With $n+1$ levels, the largest number of nodes at level l_i of the X-MIDD is $\prod_{j=1}^{i} (2k_j + 1)$. Therefore, the worst-case space complexity is $\mathcal{O}(\sum_{i=1}^{n} \prod_{j=1}^{i} (2k_j + 1))$.

It shows that the space complexity in the worst case not depend on neither the policies size, height of policy tree, nor the complexity of logical formulas in their target expressions. It only depends on the number of attributes and number of appearing attribute values in the policy tree. Similar to BDD approach, the size of the MIDD is affected heavily by the attribute ordering [1]. By implementation, we see that our algorithms have efficient performance with reasonable graph size.

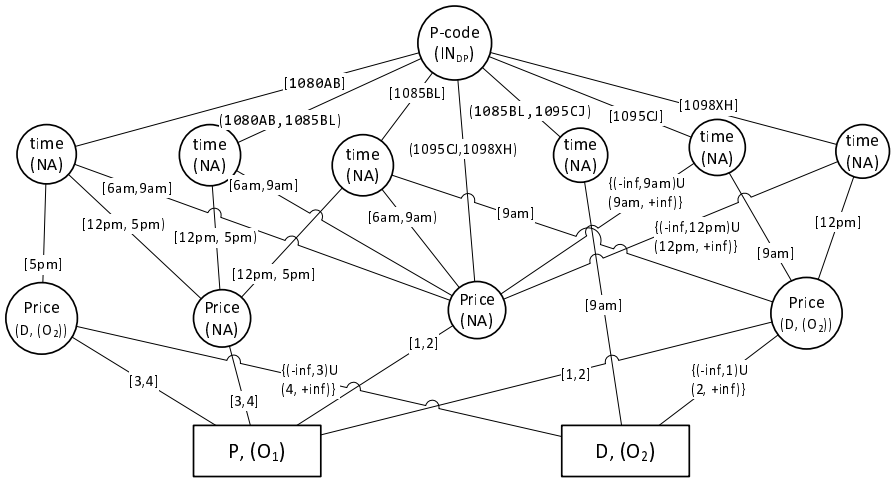

Fig. 3. X-MIDD for the sample policy

2) Evaluation time complexity: The evaluation in our X-MIDD is the traversal from the root to leaf nodes. At level l_i it has to find an applicable item among at most $2k_i+1$ out-going edges. With sequential search, the evaluation time complexity is $\mathcal{O}(\sum_{i=1}^{n}(2k_i+1))$. For binary search it is $\mathcal{O}(\sum_{i=1}^{n}\lfloor\log_2(2k_i+1)+1\rfloor)$.

Similar to space complexity, it is shown that the evaluation time complexity only depends on number of attributes n and number of appearing attribute values $k_{i|i=1..n}$. It is our advantage to evaluate large number of policies with complex logical expressions. However, the drawback is the memory cost with policies having large number of n and k_i values.

C. Experiments

In our experiments, we compare our implementation with the standard SunXACML engine [16].

We do not make direct experiments to compare with prior work [4], [5]. The work in [4] has better performance in datasets with only equality operators due to numerical comparisons are faster. In other case, it does not support datasets with complex operators. The approach in [5] does not publish neither its implementation nor datasets. However, we see that our X-MIDD structure has similar time complexity as their MT structure in the worst case. But after MT evaluation, they need to evaluate the CT, that the time complexity relies on height of policy tree. So ours has somewhat better evaluation performance.

1) Environment and datasets: We run experiments on JRE 1.7, a Linux x64 system with Intel i5 core 2.67 GHz and 4GB RAM. The experiment datasets are XACML 3.0 policies. To make it compatible with SunXACML, we have to convert to version 2.0.

We use three datasets shown in Table II. The first one is a real-life policy taken from GEYSERS project [18] with some obligations and a critical attribute marked as "MustBePresent=true". The continue-a policy is taken and converted from [11]. The synthetic-360 is our randomly generated policy using 80% equality operator and 20% other complex operators. We also choose random mixture of all combining algorithms.

TABLE II. SAMPLE POLICY DATASETS

Datasets	Policy levels	#Policy sets	#Policies	#Rules	Attributes	Operators
GEYSERS	3	6	7	33	3	=
Continue-a	6	111	266	298	14	=
Synthetic-360	4	31	72	360	10	=(80%), complex(20%)

We see that most target expressions in continue-a policy are trivial with either empty or a few match expressions in a level. The synthetic-360 has more complex expressions: each Target, AnyOf and AllOf expressions contain from 0 to 4 children. The typical target expressions thus have the complete form as in (1). It is the reason why the X-MIDD generated from synthetic-360 is more complex than from continue-a.

In our testbed, we generate requests randomly following uniformly distribution for each datasets and evaluate at two engines.

2) Validation: In our validation experiment, we randomly generate 1000 requests and compare returned decisions from two engines. Multiple experiment times on datasets confirm the correctness of our approach and implementation.

3) Performance Analysis: Fig.4 shows the average evaluation response times of a million requests for SNE-XACML and a thousand requests for SunXACML. We observe that our engine is almost linear with number of attributes. Fig.5 shows the standard deviation of evaluation response time. With all datasets, SunXACML has greater variation in response times than ours. This illustrates our analysis of evaluation time complexity, which is linear in attribute sizes and logarithmic complexity in attribute values.

Table III shows our micro-benchmark results. The preprocessing time and X-MIDD size are highly dependent on policy complexity, which is the number of attribute n and number of attribute values k. The X-MIDD of continue-a policy with 14 attributes has less nodes than one of synthetic-360 policy with 10 attributes because the later is randomly generated, thus contain higher $\{k_i\}$ values. We can see that MIDD

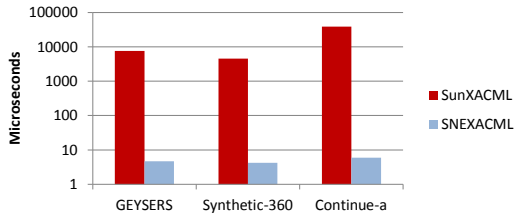

Fig. 4. Average evaluation response times

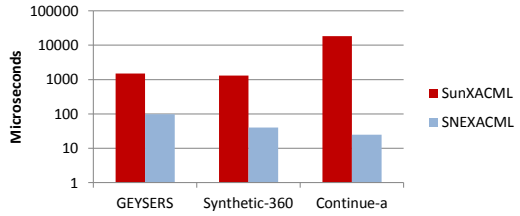

Fig. 5. Standard deviation of evaluation response times

TABLE III. MICRO-BENCHMARK EVALUATION

	GEYSERS	Continue-a	Synthetic-360
Pre-processing (ms)	94	480	1043
X-MIDD size (nodes)	55	3258	104675
Performance (requests/s)	229,551	172,114	238,878
Request conversion time fraction	14.2%	38.7%	44.8%
Response conversion time fraction	3.4%	1.6%	2.5%
MIDD evaluation time fraction	82.4%	59.7%	52.3%

evaluation time fractions are substantial and different among datasets. It illustrates that XML processing mechanisms (e.g. JAXB in our implementation) are not significant comparing to total evaluation response time. So improving the evaluation mechanism is the important factor for a high performance XACML engine.

IX. CONCLUSION

In this paper, we presented a solution to optimize XACML policy evaluation that not only gains significant performance improvement but also provides correctness and completeness features, which is missing in prior work. The proposed solution has the completeness of logic expressions defined in policies, correctness of combining algorithms semantic, critical attribute setting, obligations and advices handling. We define the MIDD tree structure which can be used both in XACML and in other attribute-based policy languages. The analysis shows that the presented solution has efficient evaluation in time complexity while having reasonable space complexity. Experiments prove that our solution implemented in the SNE-XACML engine has both the significant performance comparing to the referenced SunXACML engine and validated evaluation results.

Our future research has following directions. First we are going to integrate other XACML features [9] in the current implementation. Beside that, the MIDD operations can be utilized in policy verification redundancy detection research. In addition, we are planning to integrate our XACML engine to our concurrent work [7], [19]. Besides XACML, we also plan to support other attribute-based policy languages.

X. ACKNOWLEDGMENTS

This work is supported by the FP7 EU funded projects The Generalised Architecture for Dynamic Infrastructure Services (GEYSERS, FP7-ICT-248657), GN3plus and the Dutch national research program COMMIT.

REFERENCES

[1] R. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. C-35, no. 8, pp. 677 –691, aug. 1986.

[2] K. Strehl and L. Thiele, "Interval diagrams for efficient symbolic verification of process networks," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 19, no. 8, pp. 939 – 956, aug 2000.

[3] M. Christiansen and E. Fleury, "An MTIDD based firewall," *Telecommunication Systems*, vol. 27, pp. 297–319, 2004. [Online]. Available: http://dx.doi.org/10.1023/B%3ATELS.0000041013.23205.0f

[4] A. Liu, F. Chen, J. Hwang, and T. Xie, "Designing fast and scalable xacml policy evaluation engines," *Computers, IEEE Transactions on*, vol. 60, no. 12, pp. 1802 –1817, dec. 2011.

[5] S. Pina Ros, M. Lischka, and F. Gómez Mármol, "Graph-based XACML evaluation," in *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, ser. SACMAT '12. New York, NY, USA: ACM, 2012, pp. 83–92.

[6] Y. Demchenko, L. Gommans, and C. de Laat, "Using SAML and XACML for complex resource provisioning in grid based applications," in *Policies for Distributed Systems and Networks, 2007. POLICY '07. Eighth IEEE International Workshop on*, 2007, pp. 183–187.

[7] C. Ngo, P. Membrey, Y. Demchenko, and C. De Laat, "Security framework for virtualised infrastructure services provisioned on-demand," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, 2011, pp. 698–704.

[8] OASIS, "eXtensible Access Control Markup Language XACML version 2.0," http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[9] OASIS, "eXtensible Access Control Markup Language XACML version 3.0. OASIS standard," http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf.

[10] V. Kolovski, J. Hendler, and B. Parsia, "Analyzing web access control policies," in *Proceedings of the 16th international conference on World Wide Web*, ser. WWW '07. New York, NY, USA: ACM, 2007, pp. 677–686.

[11] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and change-impact analysis of access-control policies," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 196–205.

[12] D. Lin, P. Rao, E. Bertino, N. Li, and J. Lobo, "Policy decomposition for collaborative access control," in *Proceedings of the 13th ACM symposium on Access control models and technologies*, ser. SACMAT '08. New York, NY, USA: ACM, 2008, pp. 103–112.

[13] N. Li, Q. Wang, W. Qardaji, E. Bertino, P. Rao, J. Lobo, and D. Lin, "Access control policy combining: theory meets practice," in *Proceedings of the 14th ACM symposium on Access control models and technologies*, ser. SACMAT '09. New York, NY, USA: ACM, 2009, pp. 135–144.

[14] P. Mazzoleni, E. Bertino, B. Crispo, and S. Sivasubramanian, "XACML policy integration algorithms: not to be confused with xacml policy combination algorithms!" in *Proceedings of the eleventh ACM symposium on Access control models and technologies*, ser. SACMAT '06. New York, NY, USA: ACM, 2006, pp. 219–227.

[15] S. Marouf, M. Shehab, A. Squicciarini, and S. Sundareswaran, "Adaptive reordering and clustering-based framework for efficient xacml policy evaluation," *Services Computing, IEEE Transactions on*, vol. 4, no. 4, pp. 300 –313, oct.-dec. 2011.

[16] "Sun's XACML implementation," http://sunxacml.sourceforge.net/.

[17] D. E. Taylor, "Survey and taxonomy of packet classification techniques," *ACM Comput. Surv.*, vol. 37, no. 3, pp. 238–275, Sep. 2005.

[18] "GEYSERS - Generalised Architecture for Dynamic Infrastructure Services," http://www.geysers.eu/, 2010.

[19] M. X. Makkes, C. Ngo, Y. Demchenko, R. Strijkers, R. Meijer, and C. de Laat, "Defining intercloud federation framework for multi-provider cloud services integration," in *CLOUD COMPUTING 2013, The Forth International Conference on Cloud Computing, GRIDs, and Virtualization*, 2013, pp. 185–190.

APPENDIX A
XACML 3.0 COMBINING ALGORITHMS

TABLE IV. PERMIT-OVERRIDE ALGORITHM TRANSITION

$Q \setminus \Sigma$	P	IN_{DP}	IN_P	D	IN_D	NA
P	P	P	P	P	P	P
IN_{DP}	P	IN_{DP}	IN_{DP}	IN_{DP}	IN_{DP}	IN_{DP}
IN_P	P	IN_{DP}	IN_P	IN_{DP}	IN_{DP}	IN_P
D	P	IN_{DP}	IN_{DP}	D	D	D
IN_D	P	IN_{DP}	IN_{DP}	D	IN_D	IN_D
NA	P	IN_{DP}	IN_P	D	IN_D	NA

TABLE V. DENY-OVERRIDE ALGORITHM TRANSITION

$Q \setminus \Sigma$	P	IN_{DP}	IN_P	D	IN_D	NA
P	P	IN_{DP}	P	D	IN_{DP}	P
IN_{DP}	IN_{DP}	IN_{DP}	IN_{DP}	D	IN_{DP}	IN_{DP}
IN_P	P	IN_{DP}	IN_P	D	IN_{DP}	IN_P
D	D	D	D	D	D	D
IN_D	IN_{DP}	IN_{DP}	IN_{DP}	D	IN_D	IN_D
NA	P	IN_{DP}	IN_P	D	IN_D	NA

APPENDIX B
MIDD ALGORITHMS

Input: Two MIDDs : M_1, M_2
Output: Conjunctive join $M = M_1 \wedge M_2$

begin
  if *(some M_i is a true-leaf-node)* **then**
    return other M_j;
  end
  else if *(M_1.var = M_2.var)* **then**
    $M \leftarrow (M_1.var, C := \emptyset)$;
    $P \leftarrow M_1.P \curlywedge M_2.P$;
    foreach *(interval $I \in P$)* **do**
      $c \leftarrow conjunctive\_join(M_1.C[I], M_2.C[I])$;
      $e.P \leftarrow \{I\}$;
      $M.C \leftarrow M.C \cup (c, e)$;
    end
  end
  else
    $L \leftarrow M_i$ that has lower variable order;
    $H \leftarrow M_j$ that has higher variable order;
    $M \leftarrow (L.var, \emptyset)$;
    foreach *($(c, e) \in L.C$)* **do**
      $c' \leftarrow conjunctive\_join(c, H)$;
      $M.C \leftarrow M.C \cup \{(c', e)\}$;
    end
  end
  return M;
end

Algorithm 5: The *conjunctive_join* function

TABLE VI. FIRST-APPLICABLE ALGORITHM TRANSITION

$Q \setminus \Sigma$	P	D	NA	IN
P	P	P	P	P
D	P	D	D	D
NA	P	D	NA	IN
IN	IN	IN	IN	IN

TABLE VII. ONLY-ONE-APPLICABLE ALGORITHM TRANSITION

$Q \setminus \Sigma$	P	D	NA	IN
P	IN	IN	P	IN
D	IN	IN	D	IN
NA	P	D	NA	IN
IN	IN	IN	IN	IN

TABLE VIII. PERMIT-UNLESS-DENY ALGORITHM TRANSITION

$Q \setminus \Sigma$	P	D	NA	IN
P	P	D	P	P
D	D	D	D	D
NA	P	D	P	P
IN	P	D	P	P

TABLE IX. DENY-UNLESS-PERMIT ALGORITHM TRANSITION

$Q \setminus \Sigma$	P	D	NA	IN
P	P	P	P	P
D	P	D	D	D
NA	P	D	D	D
IN	P	D	D	D

Input: Two MIDDs : M_1, M_2
Output: Disjunctive join $M = M_1 \vee M_2$

begin
  if *(some M_i is a true-leaf-node)* **then**
    return the true-leaf-node;
  end
  else if *(M_1.var = M_2.var)* **then**
    $M \leftarrow (M_1.var, C := \emptyset)$;
    $P \leftarrow M_1.P \curlyvee M_2.P$;
    foreach *(interval $I \in P$)* **do**
      $c \leftarrow disjunctive\_join(M_1.C[I], M_2.C[I])$;
      $e.P \leftarrow \{I\}$;
      $M.C \leftarrow M.C \cup \{(c, e)\}$;
    end
  end
  else
    $L \leftarrow M_i$ that has lower variable order;
    $H \leftarrow M_j$ that has higher variable order;
    $M \leftarrow (L.var, \emptyset)$;
    foreach *($(c, e) \in L.C$)* **do**
      $c' \leftarrow disjunctive\_join(c, H)$;
      $M.C \leftarrow M.C \cup \{(c', e)\}$;
    end
  end
  return M;
end

Algorithm 6: The *disjunctive_join* function