

# Withdrawn Draft

## Warning Notice

The attached draft document has been withdrawn and is provided solely for historical purposes. It has been followed by the document identified below.

**Withdrawal Date** January 31, 2025

**Original Release Date** October 8, 2024

### The attached draft document is followed by:

**Status** Final

**Series/Number** NIST IR 8539

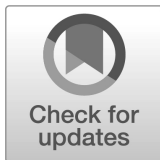
**Title** Security Property Verification by Transition Model

**Publication Date** January 2025

**DOI** <https://doi.org/10.6028/NIST.IR.8539>

**CSRC URL** <https://csrc.nist.gov/pubs/ir/8539/final>

### Additional Information



**NIST Internal Report  
NIST IR 8539 ipd**

# **Security Property Verification by Transition Model**

Initial Public Draft

Vincent C. Hu

This publication is available free of charge from:  
<https://doi.org/10.6028/NIST.IR.8539.ipd>

**NIST Internal Report**  
**NIST IR 8539 ipd**

# **Security Property Verification by Transition Model**

Initial Public Draft

Vincent C. Hu  
*Computer Security Division*  
*Information Technology Laboratory*

This publication is available free of charge from:  
<https://doi.org/10.6028/NIST.IR.8539.ipd>

October 2024



U.S. Department of Commerce  
*Gina M. Raimondo, Secretary*

National Institute of Standards and Technology  
*Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology*

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at <https://csrc.nist.gov/publications>.

### **NIST Technical Series Policies**

[Copyright, Use, and Licensing Statements](#)

[NIST Technical Series Publication Identifier Syntax](#)

### **Publication History**

Approved by the NIST Editorial Review Board on YYYY-MM-DD [Will be added in final publication.]

### **How to Cite this NIST Technical Series Publication:**

Hu VC (2024) Security Property Verification by Transition Model. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) NIST IR 8539 ipd.

<https://doi.org/10.6028/NIST.IR.8539.ipd>

### **Author ORCID iDs**

Vincent C. Hu: 0000-0002-1648-0584

### **Public Comment Period**

October 8, 2024 - November 25, 2024

### **Submit Comments**

[ir8539-comments@nist.gov](mailto:ir8539-comments@nist.gov)

National Institute of Standards and Technology  
Attn: Computer Security Division, Information Technology Laboratory  
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

### **Additional Information**

Additional information about this publication is available at <https://csrc.nist.gov/pubs/ir/8539/ipd>, including related content, potential updates, and document history.

**All comments are subject to release under the Freedom of Information Act (FOIA).**

1 **Abstract**

2 Verifying the security properties of access control policies is a complex and critical task. The  
3 policies and their implementation often do not explicitly express their underlying semantics,  
4 which may be implicitly embedded in the logic flows of policy rules, especially when policies are  
5 combined. Instead of evaluating and analyzing access control policies solely at the mechanism  
6 level, formal transition models are used to describe these policies and prove the system's  
7 security properties. This approach ensures that access control mechanisms can be designed to  
8 meet security requirements. This document explains how to apply model-checking techniques  
9 to verify security properties in transition models of access control policies. It provides a brief  
10 introduction to the fundamentals of model checking and demonstrates how access control  
11 policies are converted into automata from their transition models. The document then focuses  
12 on discussing property specifications in terms of linear temporal logic (LTL) and computation  
13 tree logic (CTL) languages with comparisons between the two. Finally, the verification process  
14 and available tools are described and compared.

15 **Keywords**

16 access control; access control policy; model test; policy test; policy verification.

17 **Reports on Computer Systems Technology**

18 The Information Technology Laboratory (ITL) at the National Institute of Standards and  
19 Technology (NIST) promotes the U.S. economy and public welfare by providing technical  
20 leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test  
21 methods, reference data, proof of concept implementations, and technical analyses to advance  
22 the development and productive use of information technology. ITL's responsibilities include  
23 the development of management, administrative, technical, and physical standards and  
24 guidelines for the cost-effective security and privacy of other than national security-related  
25 information in federal information systems.

## 26 **Call for Patent Claims**

27 This public review includes a call for information on essential patent claims (claims whose use  
28 would be required for compliance with the guidance or requirements in this Information  
29 Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be  
30 directly stated in this ITL Publication or by reference to another publication. This call also  
31 includes disclosure, where known, of the existence of pending U.S. or foreign patent  
32 applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign  
33 patents.

34 ITL may require from the patent holder, or a party authorized to make assurances on its behalf,  
35 in written or electronic form, either:

- 36 a) assurance in the form of a general disclaimer to the effect that such party does not hold  
37 and does not currently intend holding any essential patent claim(s); or
- 38 b) assurance that a license to such essential patent claim(s) will be made available to  
39 applicants desiring to utilize the license for the purpose of complying with the guidance  
40 or requirements in this ITL draft publication either:
  - 41 i. under reasonable terms and conditions that are demonstrably free of any unfair  
42 discrimination; or
  - 43 ii. without compensation and under reasonable terms and conditions that are  
44 demonstrably free of any unfair discrimination.

45 Such assurance shall indicate that the patent holder (or third party authorized to make  
46 assurances on its behalf) will include in any documents transferring ownership of patents  
47 subject to the assurance, provisions sufficient to ensure that the commitments in the assurance  
48 are binding on the transferee, and that the transferee will similarly include appropriate  
49 provisions in the event of future transfers with the goal of binding each successor-in-interest.

50 The assurance shall also indicate that it is intended to be binding on successors-in-interest  
51 regardless of whether such provisions are included in the relevant transfer documents.

52 Such statements should be addressed to [ir8539-comments@nist.gov](mailto:ir8539-comments@nist.gov)

53

54	<b>Table of Contents</b>	
55	<b>Executive Summary</b> .....	<b>1</b>
56	<b>1. Introduction</b> .....	<b>2</b>
57	<b>2. Formal Models and ACPs</b> .....	<b>3</b>
58	2.1. Model Fundamentals.....	3
59	2.2. ACP Automata.....	4
60	2.2.1. Static ACPs.....	4
61	2.2.2. Dynamic ACPs.....	5
62	2.3. ACP combinations.....	6
63	2.3.1. Nonconcurrent ACP Combinations.....	7
64	2.3.2. Concurrent ACP Combinations.....	9
65	<b>3. Properties</b> .....	<b>14</b>
66	3.1. Property Specifications.....	14
67	3.1.1. Linear Temporal Logic (LTL).....	14
68	3.1.2. Computation Tree Logic (CTL).....	15
69	3.1.3. Computation Tree Logic Star (CTL*).....	16
70	3.1.4. LTL vs. CTL (and CTL*).....	17
71	3.2. Security Properties.....	18
72	3.2.1. Safety.....	18
73	3.2.2. Liveness.....	19
74	<b>4. Verification Process</b> .....	<b>21</b>
75	4.1. General Method.....	21
76	4.2. NuSMV Tool.....	22
77	4.3. Comparison With Other Model-Checking Methods.....	22
78	<b>5. Conclusion</b> .....	<b>24</b>
79	<b>References</b> .....	<b>25</b>
80	<b>List of Tables</b>	
81	<b>Table 1. CTL vs. CTL* formulae</b> .....	<b>17</b>
82	<b>List of Figures</b>	
83	<b>Fig. 1. Example automaton of a random rules ACP</b> .....	<b>4</b>
84	<b>Fig. 2. Example automaton of a Chinese Wall ACP</b> .....	<b>5</b>
85	<b>Fig. 3. Example automaton of a Workflow ACP</b> .....	<b>6</b>

86 **Fig. 4. Example automaton of an N-person Control ACP .....6**

87 **Fig. 5. Intersection concept using an example of two automata.....7**

88 **Fig. 6. Union concept using an example of two automata.....8**

89 **Fig. 7. Concatenation concept using an example of two automata.....9**

90 **Fig. 8. Example of a combination of two interleaving automata.....10**

91 **Fig. 9. Shared variables concept with an example of two automata .....11**

92 **Fig. 10. Shared actions concept with an example of two automata .....12**

93 **Fig. 11. Example of  $E(EX p) U (AG q)$  in CTL.....16**

94 **Fig. 12. Relationships among LTL, CTL, and CTL\* .....18**

95 **Fig. 13. Example of the ACP transition model that satisfies  $EG p$  but not  $AF q$  .....21**

96 **Fig. 14. A mutual exclusion access system.....21**

97



98 **Acknowledgments**

99 The author would like to express his thanks to Isabel Van Wyk and Jim Foti of NIST for their  
100 detailed editorial review of both the public comment version and the final publication.

## 101 **Executive Summary**

102 Faults may be errors or weaknesses in the design or implementation of access control policies  
103 that can lead to serious vulnerabilities. This is particularly true when different access control  
104 policies are combined. The issue becomes increasingly critical as systems grow more complex,  
105 especially in distributed environments like the cloud and IoT, which manage large amounts of  
106 sensitive information and resources that are organized into sophisticated structures. Verifying  
107 the security properties of access control policies is a complex and critical task. The policies and  
108 their implementation often do not explicitly express their underlying semantics, which may be  
109 implicitly embedded in the logic flows of policy rules, especially when policies are combined.

110 Formal transition models are used to prove the policy's security properties and ensure that  
111 access control mechanisms are designed to meet security requirements. This report explains  
112 how to apply model-checking techniques to verify security properties in transition models of  
113 access control policies. It provides a brief introduction to the fundamentals of model checking  
114 and demonstrates how access control policies are converted into automata from their  
115 transition models. The report then focuses on discussing property specifications in terms of  
116 linear time logic (LTL) and computation tree logic (CTL) with comparisons between the two.  
117 Finally, the verification process and available tools are described and compared.

118

## 119 **1. Introduction**

120 Faults can lead to serious vulnerabilities, particularly when different access control policies  
121 (ACPs) are combined. This issue becomes increasingly critical as systems grow more complex,  
122 especially in distributed environments like the cloud and IoT, which manage large amounts of  
123 sensitive information and resources that are organized into sophisticated structures. NIST  
124 Special Publication (SP) 800-192 [SP192] provides an overview of ACP verification using the  
125 model-checking method. However, it does not formally define the automata of transition  
126 models and properties, nor does it detail the processes and considerations for verifying access  
127 control security properties.

128 Instead of evaluating and analyzing ACPs solely at the mechanism level, formal models are  
129 typically developed to describe their security properties. An ACP transition model is a formal  
130 representation of the ACP as enforced by the mechanism and is valuable for proving the  
131 system's theoretical limitations. This ensures that access control mechanisms are designed to  
132 adhere to the properties of the model. Generally, transition models are effective for modeling  
133 non-discretionary ACPs.

134 An automaton is an abstraction of a self-operating transition model that follows a  
135 predetermined sequence of operations or responses. To formally verify the properties of ACP  
136 transition models through model checking, these models need to be converted into automata.  
137 This allows the rules of the ACP to be represented as a predetermined set of instructions within  
138 the automaton.

139 This document explains model-checking techniques for verifying access control security  
140 properties using the automata of ACP transition models. It briefly introduces the fundamentals  
141 of model checking and demonstrates how access control policies are converted into automata  
142 from transition models. The document then delves into discussions of property specifications  
143 using linear temporal logic (LTL) and computation tree logic (CTL) languages with comparisons  
144 between the two. The process of verification and the available tools are also described and  
145 compared. This document is organized as follows:

- 146 • Section 1 is the introduction.
- 147 • Section 2 provides an overview of formal models and ACPs.
- 148 • Section 3 describes properties.
- 149 • Section 4 explains the property verification process.
- 150 • Section 5 is the conclusion.
- 151 • The References section lists cited publications and sources.

152

## 153 2. Formal Models and ACPs

154 This section explains the application of formal models to ACPs.

### 155 2.1. Model Fundamentals

156 With general computational systems, one method to formally verify the properties of an ACP is  
157 to apply model checking. This process begins by describing the ACP as a transition model and  
158 converting it into a system of automata, which are mathematical structures used to represent  
159 and analyze the behavior of computational systems. The automata deal with the logic of  
160 computation concerning the ACP transition systems and include various types, such as finite  
161 automata, Büchi automata, pushdown automata, Turing machines automata, linear bounded  
162 automata, and cellular automata. Both finite and Büchi automata have deterministic and  
163 nondeterministic types (i.e., DFA, NFA, DBA, and NBA).

164 In static and dynamic ACPs, each access control rule must lead to only one access state. There is  
165 only one permission result for each access request, which means that there are no  
166 nondeterministic state transitions in ACP automata. Additionally, there is generally no  
167 requirement for in-state memory in ACPs, making DFA and DBA sufficient to express ACP  
168 transition systems for most models, such as attribute-based access control (ABAC), role-based  
169 access control (RBAC), workflow management, separation of duties (SOD), conflict of interest  
170 (COI), and N-person control [SP162]25. The following are some common features of automata  
171 applied to ACP models:

- 172 • To represent the rules of an ACP, a deterministic automaton has a finite number of  
173 states, and each state has a unique deterministic transition for every access control  
174 request input or system action. This automaton is used to recognize security properties  
175 that are specified in the temporal logic of regular languages. In contrast,  
176 nondeterministic automata can have multiple transitions for a given input symbol,  
177 including transitions to multiple states or no states at all. Therefore, they are not  
178 applicable to ACPs, even though nondeterministic automata are also used to recognize  
179 regular languages.
- 180 • An ACP may require monitoring the current state continuously, so automata must be  
181 capable of handling infinite sequences of inputs. Such automata are called Büchi  
182 automata (BA), which are designed to determine whether a language is accepted in  
183 infinite words. A word is accepted by a BA if there is a run in which some accepting state  
184 occurs infinitely often. In contrast to finite automata (FA), which accept finite words that  
185 must end in an accepting state, BA can accept infinite words as long as there is a run (or  
186 trace) of the automaton that passes through an accepting state.
- 187 • Some ACPs may be constructed using “deny” conditions instead of “grant” conditions. In  
188 such cases, they can utilize the complement of DFA language by switching accepting  
189 states to non-accepting states and vice versa for the ACP transition models. However,  
190 this feature applies only to DFA and not to BA.

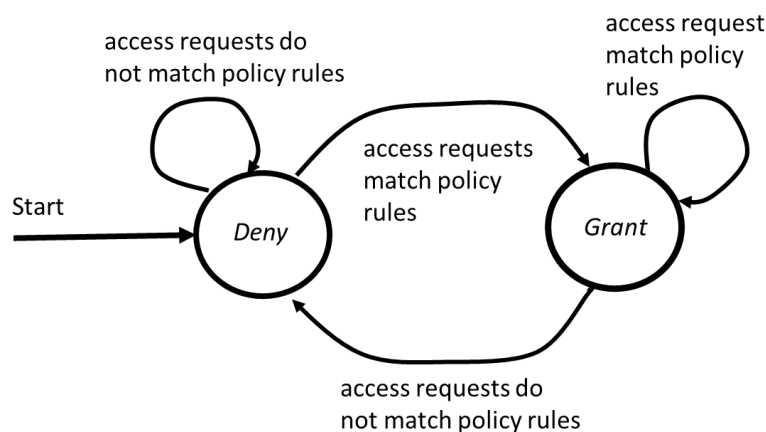
- 191 • To evaluate access permissions, the automaton’s accepting states represent either the  
192 grant or deny permissions of ACP, depending on the default setting and specific system  
193 actions. If an input move triggered by an access request cannot reach an accepting state  
194 (e.g., grant, deny, or specific system actions), it indicates that the request is invalid. In  
195 this case, the request is assigned the default permission while the automaton remains in  
196 the same state.
- 197 • An “empty automaton” typically refers to a type of finite automaton that does not  
198 accept any input string nor recognizes any language, including the empty string. An  
199 empty automaton signifies an ACP that blocks all valid access requests. Therefore, if a  
200 valid input access request (e.g., the subjects, actions, and objects of the request are  
201 recognizable) does not transition to any accepting state (e.g., grant, deny, or other  
202 acceptable states), it indicates that the automaton does not correctly represent the ACP.

## 203 2.2. ACP Automata

204 This section illustrates how static and dynamic ACP models are translated into automata.

### 205 2.2.1. Static ACPs

206 Static ACPs regulate access permissions based on static system states that are defined by  
207 conditions, such as attribute propositions and system environments (e.g., time, location, etc.).  
208 Some popular static ACP models include access control lists (ACLs), ABAC, and RBAC, where the  
209 ACPs are typically defined by rules that specify access control variables, including subjects,  
210 actions, objects, and environmental conditions. These ACPs are specified by independent (i.e.,  
211 asynchronous) states within an automaton. A current state/next state pair is only included in  
212 the transition relation if it satisfies the ACP rule variables, as illustrated in the example of  
213 random ACP rules shown in Fig. 1.



214

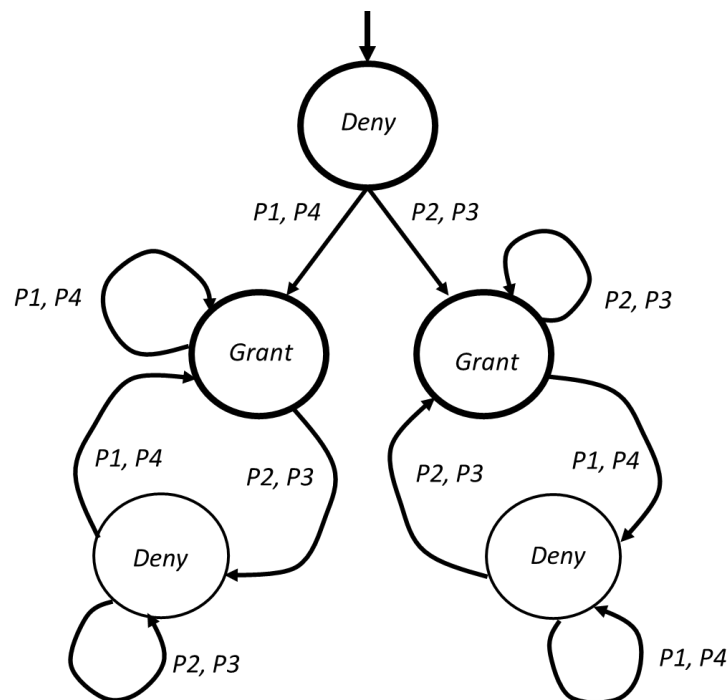
215 **Fig. 1. Example automaton of a random rules ACP**

216 In the automaton, the access authorization state is initialized as the deny state and transitions  
217 to the grant state for any access request that complies with the rule’s constraints. Otherwise, it

218 remains in the deny state. Even though environmental condition variables (e.g., time, location)  
219 may change through monitoring, they do not affect state transitions from the perspective of  
220 the automaton.

### 221 2.2.2. Dynamic ACPs

222 Dynamic ACPs regulate access permissions based on temporal constraints or conditions, such as  
223 specified events triggered by permitted access or system counters/variables controlled and/or  
224 monitored by the ACP. The automata typically contain more than one accepting state. An  
225 example of a dynamic ACP is the Chinese Wall model, which enforces a conflict-of-interest  
226 property. For instance, if *Subject 1* accesses *Object X*, then *Subject 2* is not allowed to access the  
227 same object, as illustrated in the automaton shown in Fig. 2.



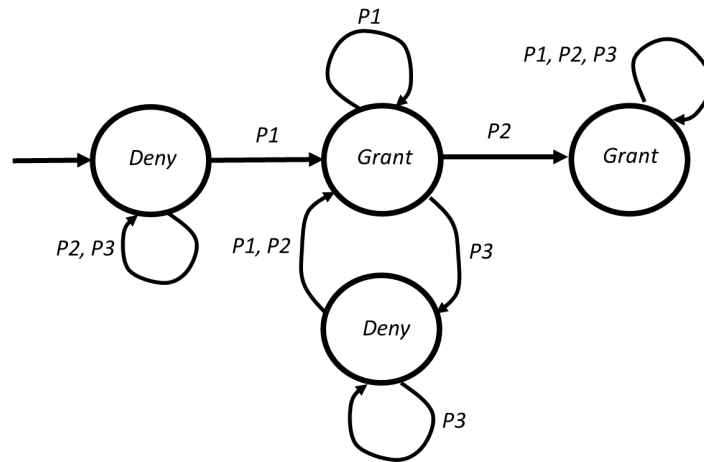
228

229

Fig. 2. Example automaton of a Chinese Wall ACP

230 Valid access requests in this scenario include *P1*: *Subject 1* accesses *Object X*, *P2*: *Subject 2*  
231 accesses *Object Y*, *P3*: *Subject 1* accesses *Object Y*, and *P4*: *Subject 2* accesses *Object Y*.

232 Figure 3 presents an automaton for the Workflow ACP model, where access  $P3$  is only allowed  
233 after  $P2$ , and access  $P2$  is only allowed after  $P1$ .

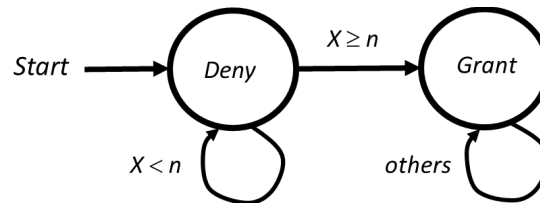


234  
235

Fig. 3. Example automaton of a Workflow ACP

236 Figure 4 depicts an automaton for an N-person control ACP model in which an object can only  
237 be accessed when the access count  $X$  exceeds a specified value  $n$ .

238



239  
240

Fig. 4. Example automaton of an N-person Control ACP

### 241 2.3. ACP combinations

242 An ACP is not necessarily expressed by a single model. It can also be implicitly embedded by  
243 being mixed with other ACP models or a random set of access rules. Consequently, an ACP  
244 automaton may be represented by combining multiple ACP automata or adding constraint  
245 states outside of ACP models. Such a combined ACP must concurrently manage access to  
246 achieve the unified access control behavior that results from the incorporation or federation of  
247 multiple ACPs or rules. In practical applications, for instance, a local system's ACP may need to  
248 be integrated with a global ACP (or meta ACP) in a distributed system environment, such as in  
249 cloud computing or IoT devices with centralized access control.

250 In general, ACP combinations can be divided into two categories: nonconcurrent and  
251 concurrent combinations. A nonconcurrent combination does not require synchronization  
252 between the combined ACPs, while a concurrent combination does need to account for the  
253 synchronization of shared states or variables between the ACPs. To ensure that the combined  
254 ACP functions correctly, it is essential to formally detect any inconsistencies or incompleteness,  
255 such as scenarios in which an access request is both accepted and denied or where the request

256 is neither accepted nor denied according to the combined automata [SP192]. Each type of  
257 combination has distinct characteristics.

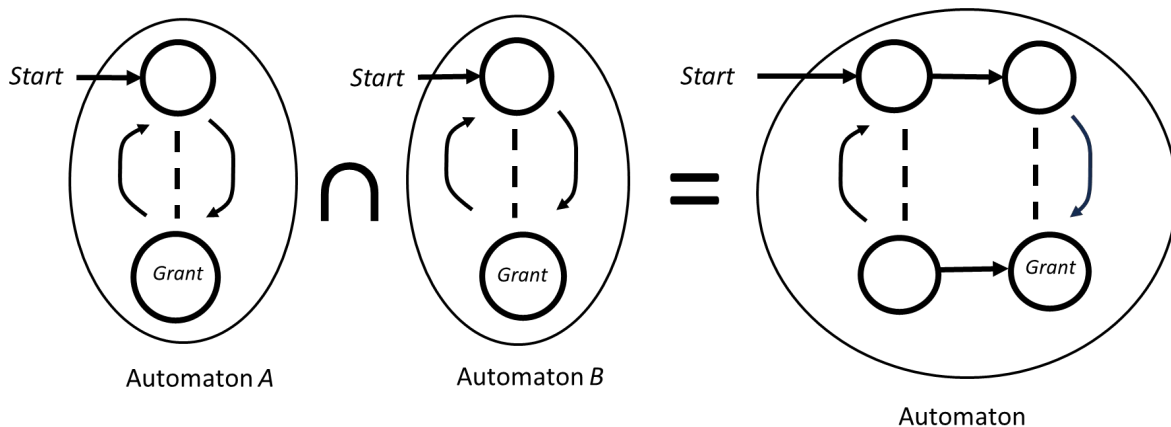
### 258 2.3.1. Nonconcurrent ACP Combinations

259 In general, nonconcurrent ACP combinations include intersection, union, and concatenation.  
260 These ACPs can be combined offline before the system’s authorization process is executed.

#### 261 2.3.1.1. Intersection

262 An access control system that requires its resources to be managed by different ACPs with  
263 common control elements (e.g., subjects, objects, or environmental conditions) means that  
264 organizations  $O_1, O_2, \dots, O_n$  have equal authority over a shared resource. To access this  
265 resource, an access request must be granted by each organization, which necessitates the use  
266 of an intersection automaton.

267 Let  $A_i$  represent the automaton of  $ACP_i$  for organization  $O_i$ . The intersection of automata  $A_1,$   
268  $A_2, \dots, A_n$  is denoted as  $A_1 \cap A_2 \dots \cap A_n$ , which forms a new automaton in which the set of states,  
269 transition functions, initial states, and accepting states are defined such that it accepts a “string  
270 of access requests” (“string” for brevity) if and only if  $A_1, A_2, \dots, A_n$  accept it. In other words, the  
271 new automaton recognizes the “language that represents all possible access request  
272 sequences” (“language” for brevity) that contains only those strings accepted by  $A_1, A_2, \dots, A_n$ . A  
273 string is accepted by the intersection automaton if and only if it is accepted by the behavior of  
274 all original automata. The intersection operation requires the Cartesian product (i.e., dot  
275 product) of the state spaces of  $A_1, A_2, \dots, A_n$ :  $A_1 \times A_2 \dots \times A_n$  (or  $A_1 \bullet A_2 \dots \bullet A_n$ ). This means that the  
276 resulting automaton consists of states that combine the states from  $A_1, A_2, \dots, A_n$ . The  
277 intersection operation focuses on the common language recognized by all automata, while the  
278 product operation emphasizes the combination of the behaviors of the automata. Figure 5  
279 illustrates the concept of intersection using an example of two automata:  $A \cap B$ .



280

281

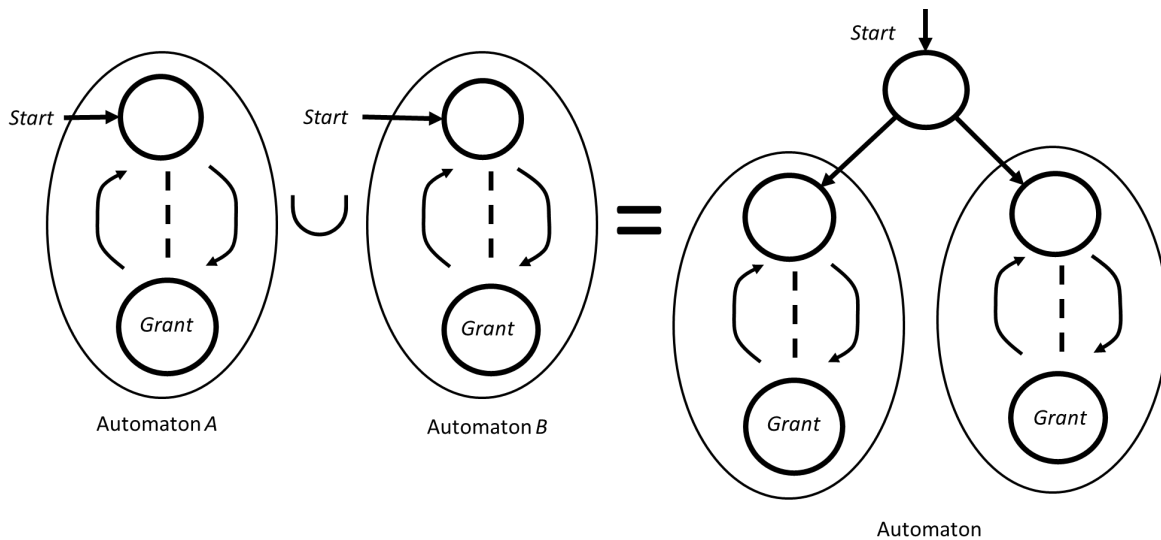
Fig. 5. Intersection concept using an example of two automata



282 **2.3.1.2. Union**

283 An access control system allows any subject from the joined systems to access the resources  
284 managed by any of them (e.g., coupon discounts offered to all customers of allied businesses).  
285 This union operation can be implemented using a union automaton, which simply merges  
286 multiple automata into a single one and enables each automaton to operate independently  
287 without consulting the others.

288 For example, let  $A_i$  represent the automaton of  $ACP_i$ , which is the ACP of the joined business  $B_i$ .  
289 The union of automata  $A_1, A_2, \dots, A_n$  is denoted as  $A_1 \cup A_2 \dots \cup A_n$  and forms a new automaton  
290 in which the set of states, transition functions, initial state, and accepting states are defined and  
291 operate independently in each  $A_x$ . An initial state is added to the union of the automata to  
292 accept all access requests and determine which  $A_x$  should handle an input access request.  
293 Figure 6 illustrates the union concept with an example of two automata:  $A \cup B$ .



294

295

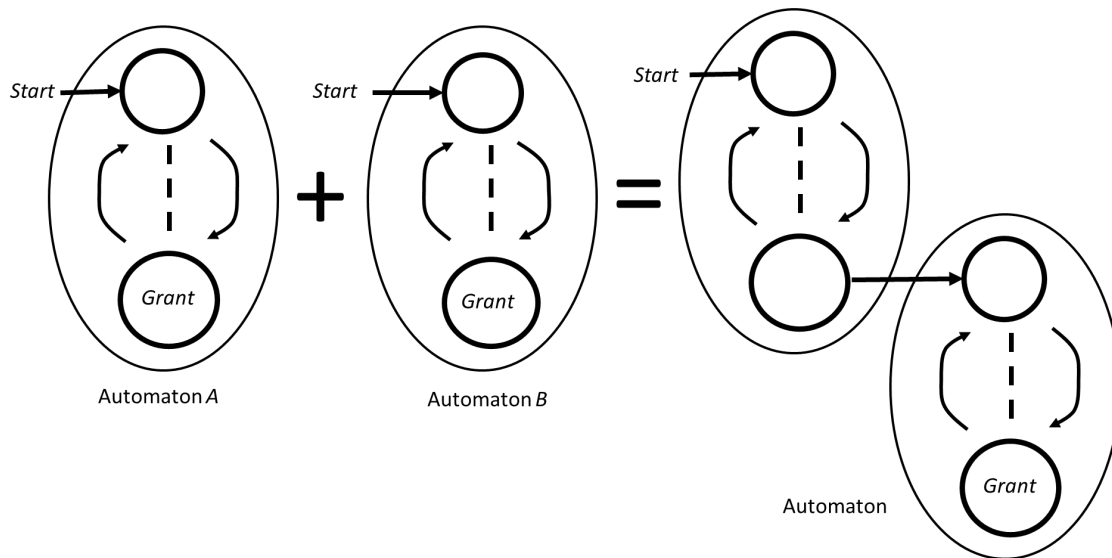
Fig. 6. Union concept using an example of two automata

296 **2.3.1.3. Concatenation**

297 Access control systems that manage the sequence of user requests or processes should  
298 consider concatenating ACPs for workflow operations. For instance, this approach is essential in  
299 an assembly line that requires approval in a predefined sequence by different work units, each  
300 of which has its own unique ACP.

301 Let  $A_i$  represent the automaton of  $ACP_i$ , which corresponds to work unit  $W_i$ . The concatenation  
302 of automata  $A_1, A_2, \dots, A_n$  involves connecting these automata end to end such that the output  
303 of the first automaton serves as the input to the next. Formally,  $A_1 + A_2 \dots + A_n$  denotes the  
304 automaton that results from linking the accepting states of  $A_1$  to the initial state of  $A_2$ , the  
305 accepting states of  $A_2$  to the initial state of  $A_3$ , and so on. This process effectively creates a new  
306 automaton that recognizes strings accepted first by  $A_1$ , which then pass through the

307 subsequent concatenated automata until reaching  $A_n$ . Figure 7 illustrates the concept of  
308 concatenation with an example of two automata:  $A + B$ .



309  
310

Fig. 7. Concatenation concept using an example of two automata

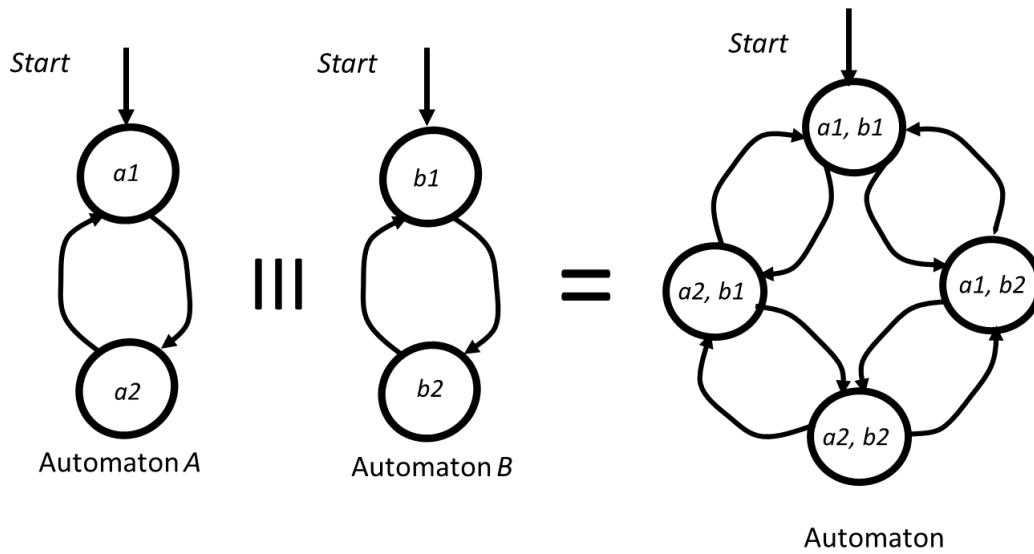
### 311 2.3.2. Concurrent ACP Combinations

312 Concurrent (i.e., interleaving or parallel) ACP combinations can be represented by an  
313 automaton in which each ACP automaton signifies a process or component, and transitions  
314 between states represent possible actions or events that can occur in parallel. These automata  
315 typically operate concurrently, allowing multiple AC authorization processes to be executed  
316 simultaneously, similar to multi-threaded programs, distributed systems, and hardware circuits.  
317 Analyzing concurrent automata can provide insights into the authorization processes of ACPs  
318 when they interact and address related issues, such as race conditions, deadlocks, and  
319 communication protocols. Generally, concurrent automata involve independent and shared  
320 variables as well as shared actions. This type of combination is performed online while  
321 authorization is in progress.

#### 322 2.3.2.1. Independent

323 Some situation-awarded ACPs rely on synchronized states to determine access permissions  
324 (e.g., air traffic control systems monitor multiple runway situations to manage access to  
325 runways). Such systems can employ independent concurrent automata for safety checks,  
326 similar to a traffic light system that only permits specific light combinations. The interleaving of  
327 independent systems operates in a way that allows their states to change dynamically and  
328 interleave with one another, meaning that the authorization processes run independently and  
329 disregards the order in which they are executed. This type of combination is a variant of the  
330 nonconcurrent intersection combinations (see Sec. 2.3.1). Each ACP has its own set of  
331 environment variables so that instead of sharing variables or actions, they share system states.  
332 Formally,  $A_1 \parallel A_2 \dots \parallel A_n$ , where  $A_i$  represents the automaton of  $ACP_i$  for concurrent access

333 control system  $S_i$ 's ACP, and the symbol III is the interleaving operator. Figure 8 illustrates the  
334 concept of independent concurrency with an example of two automata:  $A \text{ III } B$  [BK].



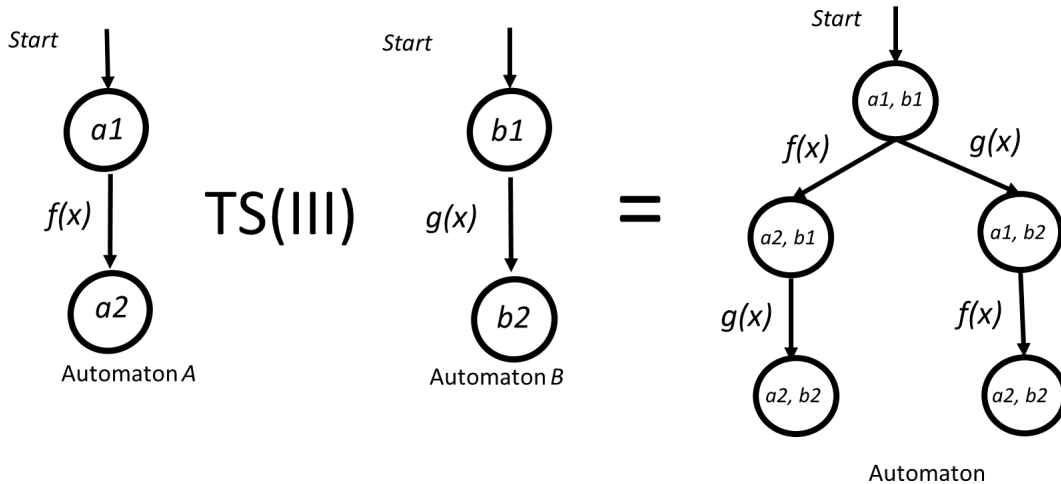
335

336

Fig. 8. Example of a combination of two interleaving automata

### 337 2.3.2.2. Shared Variables

338 For ACPs that require shared variable control (e.g., n-person control, mutual exclusion, and SOD  
339 models), their shared variables (e.g., the number of accesses or indicators of current access  
340 states) are essential for authorization processes. If these shared global variables are managed  
341 by automata in an independent concurrent combination (as described in an independent  
342 combination) and each ACP is permitted to modify them, then conflicts may arise that lead to  
343 inconsistent results for the same variables. Therefore, the concurrent automaton for shared  
344 variables must incorporate change actions as inputs for state transitions rather than merely  
345 interleaving states. Formally, this can be represented as  $TS(S_1 \text{ III } S_2 \dots \text{ III } S_2)$  instead of  $S_1 \text{ III } S_2 \dots$   
346  $\text{III } S_n$ , where  $TS$  represents the transition model. Figure 9 illustrates the concept of shared  
347 variables with an example of two automata, where  $x$  is a shared variable and  $f(x)$  and  $g(x)$  are  
348 actions that modify  $x$ .



349

350

**Fig. 9. Shared variables concept with an example of two automata**

351 A common example of a shared variable combination automaton is the enforcement of a  
 352 limited number of concurrent accesses to an object. In this case, the authorization process for a  
 353 subject consists of four states: idle, entering, critical, and exiting. The subject typically starts in  
 354 the idle state. When the user requests access to the critical object, the subject transitions to the  
 355 entering state. If the limit on concurrent access has not been reached, the subject then moves  
 356 to the critical state, and the current access count is incremented by 1. Once the subject finishes  
 357 accessing the critical object, it transitions to the exiting state, and the current access count is  
 358 decremented by 1. Finally, the subject moves from the exiting state back to the idle state. The  
 359 shared variable automaton can be modeled in the following example in pseudocode [SP192].

```

360 { VARIABLES
361   count, access_limit : INTEGER;
362   request_1 : process_request (count);
363   request_2 : process_request (count);
364   .....
365   request_n : process_request (count);
366   /*max number of user requests allowed by the system*/
367   access_limit := k; /*max number of concurrent access*/
368   count := 0; act {rd, wrt}; object {obj};
369   process_request (access_limit) {
370     VARIABLES
371     permission : {start, grant, deny};
372     state : {idle, entering, critical, exiting};
373     INITIAL_STATE (permission) := start;
374     INITIAL_STATE (state) := idle;
375     NEXT_STATE (state) := CASE {
376       state == idle : {idle, entering};
377       state == entering & ! (count > access_limit): critical;
378       state == critical : {critical, exiting};
    
```

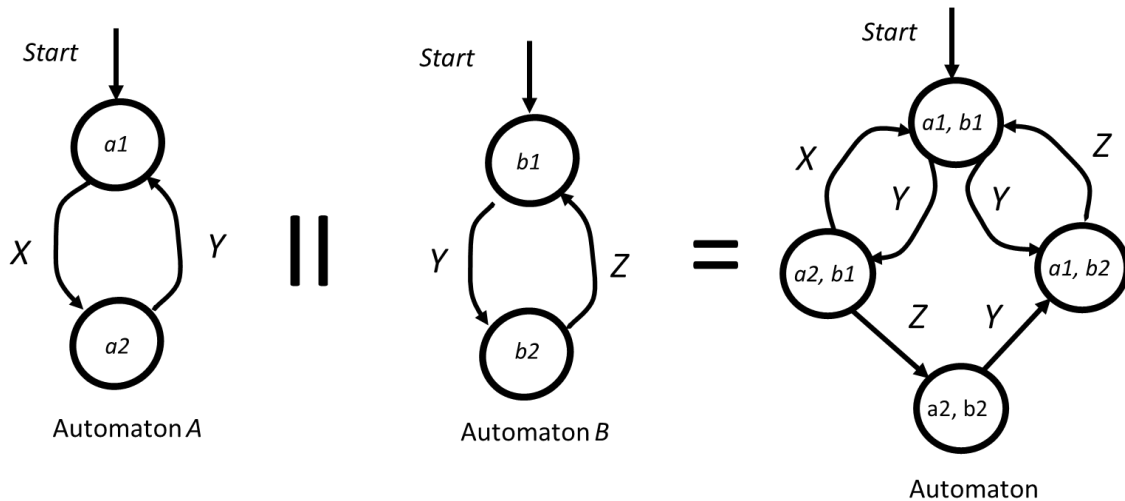
```

379         state == exiting : idle;
380         OTHERWISE: state};
381     NEXT_STATE (count) := CASE {
382         state == entering : count + 1;
383         state == exiting : count -1;
384         OTHERWISE: DO_NOTHING };
385     NEXT_STATE (permission) := CASE {
386         (state == entering)& (act == rd) & (object == obj): grant;
387         OTHERWISE: deny;};
388     }
389 }
    
```

390 **2.3.2.3. Shared actions**

391 In some ACPs, the authorization process requires a “handshaking” between systems. These  
 392 handshakes are initiated by the results of permitted actions on objects that are managed by  
 393 other systems. Shared actions in concurrent systems reflect the behavior of these handshake  
 394 actions between the states of different systems.

395 Shared actions automata are similar to concatenated automata. However, the former operates  
 396 concurrently rather than sequentially. This concurrent combination of shared actions is typically  
 397 applied to policy-based access control (PBAC) models in which permission decisions are  
 398 dynamically made based on the context of the actions of each combined ACP. Formerly, let  $A_i$   
 399 represent the automaton of  $ACP_i$  for the shared action system  $S_i$ 's ACP. The shared automaton  
 400 is formally expressed as  $A_1 \parallel A_2 \dots \parallel A_n$ , where  $\parallel$  denotes the handshake operator. Figure 10  
 401 illustrates the concept of shared actions with an example of two automata, where  $X$ ,  $Y$ , and  $Z$   
 402 are actions, and  $Y$  is the shared action.



403

404

Fig. 10. Shared actions concept with an example of two automata

405 Concurrent automata are constructed from multiple transition models of ACPs. An accepting  
 406 state (e.g., grant or deny) of the combined automata must be one of the combinations of the

407 individual accepting states from all of the automata. In the worst-case scenario, for  $n$  ACPs, the  
408 maximum number of states of the combined automata is  $O(2^n)$ .  
409

### 410 3. Properties

411 Properties are typically expressed as logical propositions that are constrained by path  
412 quantifiers or temporal conditions. They are used to verify whether they hold true throughout  
413 the transition model, thereby ensuring that certain critical aspects of system behavior remain  
414 consistent across different states or executions of the system.

#### 415 3.1. Property Specifications

416 To verify a transition model using automata, property statements (i.e., propositions expressed  
417 in Boolean functions) are supplemented with constraints or terms that define system behavior.  
418 Generally, properties can be specified in three ways:

- 419 1. Path quantifiers or temporal operators, such as U, G, F, and X
- 420 2. Finite automata
- 421 3. Regular expressions, including  $\omega$ -regular expressions

422 While these three methods can be mathematically transformed into one another, it is often  
423 more intuitive, efficient, and expressive to use path quantifiers and temporal operators to  
424 specify access control security properties. Therefore, without a loss of generality, this  
425 document will focus solely on using path quantifiers and temporal operators to demonstrate  
426 property verifications in two categories of languages: LTL and CTL.

##### 427 3.1.1. Linear Temporal Logic (LTL)

428 Linear temporal logic (LTL) [NU] is a formal language used to specify and reason about the  
429 behavior of systems over time, particularly in the fields of model checking and formal  
430 verification. It is often used in model-checking algorithms that operate on transition models or  
431 Kripke structures to verify temporal properties. LTL describes system behavior over linear time,  
432 meaning that it considers a single path of execution of events or states within the system. It  
433 employs temporal operators as a formalism to specify how the properties of the system evolve  
434 over time, thus forming a comprehensive logical framework.

435 In LTL, Boolean operators are used to specify transition states and path formulas, including  
436 negation ( $\neg$ ), which represents logical NOT; conjunction ( $\wedge$ ), which represents logical AND;  
437 disjunction ( $\vee$ ), which represents logical OR; implication ( $\rightarrow$ ), which represents logical  
438 implication; and biconditional ( $\leftrightarrow$ ), which represents logical equivalence. Common temporal  
439 operators in LTL include:

- 440 • X (Next):  $Xp$  means  $p$  holds in the next state.
- 441 • F (Eventually, finally, or somewhere):  $Fp$  means  $p$  will hold at some point in the future.
- 442 • G (Globally or always):  $Gp$  means  $p$  holds at every point in the future.
- 443 • U (Until):  $pUq$  means  $p$  holds until  $q$  holds, where  $p$  and  $q$  are properties (e.g., in  
444 Boolean propositions).

445 For example,  $GFp$  (infinitely often) means that  $Fp$  is true at infinitely many points along the  
446 trace.  $FGp$  (eventually forever) indicates that  $Gp$  will be true at some point in the future and will  
447 remain true thereafter. An LTL expression can be a combination of temporal operations and  
448 propositional logic, such as  $F(\neg p_1 \wedge X(\neg p_2 \cup p_1))$ .

### 449 3.1.2. Computation Tree Logic (CTL)

450 Computation tree logic (CTL) [NU] is a formal language used for specifying and reasoning about  
451 system behavior through a tree representation of the transition model, particularly in the fields  
452 of model checking and formal verification. It describes overall events or states over branching  
453 time, meaning that it considers multiple paths of system execution simultaneously.

454 In contrast to LTL, which does not use path quantifiers in its state formulas and focuses solely  
455 on a single path of execution, CTL utilizes Boolean operators in conjunction with path  
456 quantifiers and temporal operators to construct logical formulas to describe properties across  
457 multiple paths. The main path quantifiers in CTL are:

- 458 • A (For all):  $A p$  means  $p$  holds for all paths (tree branches) starting from the current  
459 state.
- 460 • E (There exists):  $E p$  means there exists at least one path (tree branch) starting from the  
461 current state where  $p$  holds, where  $p$  and  $q$  are properties (e.g., in Boolean  
462 propositions).

463 CTL combines these path quantifiers with LTL temporal operators. Some common combinations  
464 include:

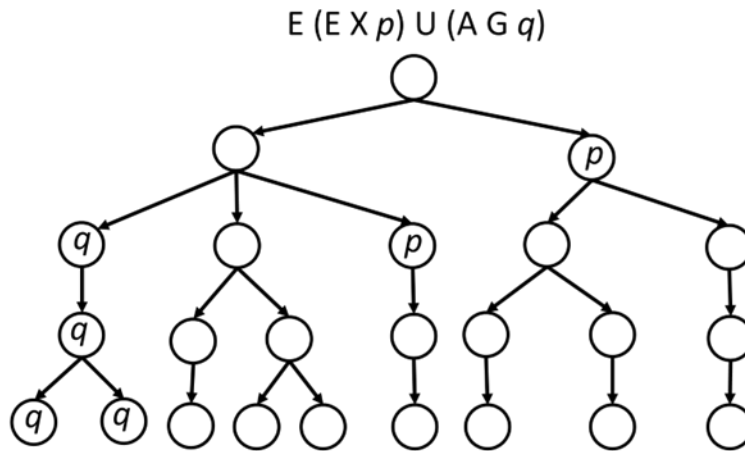
- 465 • AX (For all next):  $AX p$  means  $p$  holds in all next states.
- 466 • EX (Exists next):  $EX p$  means that there exists a next state where  $p$  holds.
- 467 • AF (For all future):  $AF p$  means  $p$  will eventually hold on all paths.
- 468 • EF (Exists future):  $EF p$  means there exists a path where  $p$  will eventually hold.
- 469 • AG (For all globally):  $AG p$  means  $p$  holds globally on all paths.
- 470 • EG (Exists globally):  $EG p$  means there exists a path where  $p$  holds globally.
- 471 •  $A(p U q)$ : Means  $p$  holds until  $q$  holds on all paths.
- 472 •  $E(p U q)$ : Means there exists a path where  $p$  holds until  $q$  holds.

473 CTL is usually expressed in a formula that uses path quantifiers to modify proposition logic and  
474 other path quantifiers. For example:

- 475 •  $AG(p \rightarrow \neg q)$ : For all paths globally, if  $p$  is true, then  $q$  is not true.
- 476 •  $E(p \vee q) U r$ : There exists a path where  $p$  or  $q$  holds until  $r$  holds.



477 Figure 11 shows an example of  $E(EX p) U (AG q)$  [YC].



478  
 479 Fig. 11. Example of  $E(EX p) U (AG q)$  in CTL

480 **3.1.3. Computation Tree Logic Star (CTL)**

481 Computation tree logic star (CTL\*) is an extension of CTL that allows for more flexible  
 482 combinations of path quantifiers and temporal operators, including nested temporal  
 483 modalities. This extension leads to more complex and expressive formulas compared to CTL.  
 484 The key differences include:

- 485 • In CTL, path quantifiers (A, E) must be immediately followed by temporal operators (X, F,  
 486 G, U). In contrast, CTL\* does not impose this restriction, and path quantifiers can be  
 487 used without an immediate temporal operator. Consequently, formulas in CTL\* can be  
 488 either state formulas or path formulas. State formulas are evaluated at individual states,  
 489 while path formulas are evaluated over paths. This means that in CTL, each X, U, F, and  
 490 G can only have one associated E or A, whereas CTL\* does not have this limitation.
- 491 • Unlike CTL, which uses Boolean operators solely for state formulas, CTL\* allows for the  
 492 combination of both state and path formulas. For instance, it can express properties  
 493 such as “There exists a path where, globally, some condition holds until another  
 494 condition is satisfied,” which is represented as  $AG(Fp \rightarrow EXq)$ . This means that along all  
 495 paths globally, if  $p$  eventually holds, then  $q$  must hold in the next state.
- 496 • CTL does not allow for the negation of the path formula (e.g.,  $\neg E \neg (p \cup q)$ ) but CTL\*  
 497 does.

498 The differences between CTL and CTL\* are defined by their grammar, as outlined below:

- 499 • CTL grammar
  - 500 ○ State formulae:  $\phi := \text{true} | p_i | \phi_1 \wedge \phi_2 | \neg \phi_1 | E \alpha | A \alpha$
  - 501 ○ Path formulae:  $\alpha := X \phi_1 | \phi_1 U \phi_2 | F \phi_1 | G \alpha_1$
- 502 • CTL\* grammar

- 503           ○ State formulae:  $\phi := \text{true} \mid p_i \mid \phi_1 \wedge \phi_2 \mid \neg \phi_1 \mid E \alpha \mid A \alpha$
- 504           ○ Path formulae:  $\alpha := \phi \mid \alpha_1 \wedge \alpha_2 \mid \neg \alpha_1 \mid X \alpha_1 \mid \alpha_1 U \alpha_2 \mid F \alpha_1 \mid G \alpha_1$ , where  $\phi$ ,  $\phi_1$
- 505           and  $\phi_2$  are state formulae, and  $\alpha$ ,  $\alpha_1$  and  $\alpha_2$  are path formulae

506 Table 1 shows comparisons of CTL and CTL\* formulae examples [YC2].

507 **Table 1. CTL vs. CTL\* formulae**

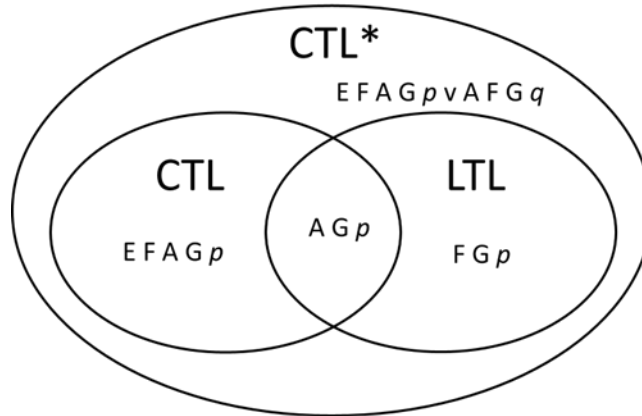
Legal CTL formulae	CTL* (illegal CTL) formulae
$E F p$	$A F G p$
$E F A G p$	$E G F p$
$A X p$	$A p$
$A F p \wedge A G q$	$A (F p \wedge G q)$
$A (p U (E G q))$	$A (p U (G q))$

508 CTL is a subset of CTL\* and is easier to use and more efficient in terms of model-checking  
 509 algorithms but also less expressive. While CTL cannot express all of the properties that CTL\*  
 510 can, it provides a good balance of expressiveness and efficiency for many practical applications.  
 511 In contrast, CTL\* offers greater expressiveness but comes with increased complexity in model  
 512 checking [HR].

### 513 3.1.4. LTL vs. CTL (and CTL\*)

514 Some properties that are expressible in LTL may involve temporal operators that cannot be  
 515 directly translated into CTL due to its branching nature. For example, LTL can more naturally  
 516 express properties like “the next state satisfies property  $p$  until property  $q$  is satisfied.” Notably,  
 517 CTL\* can express a broader range of properties compared to CTL, including some that resemble  
 518 those that are expressible in LTL. Compared to LTL, CTL\* allows for nested temporal modalities  
 519 and the use of Boolean connectives at the top level of the formula, which enhances its  
 520 expressiveness and capability to capture complex temporal behaviors.

521 From a complexity perspective, although CTL\* encompasses both CTL and LTL, CTL algorithms  
 522 are generally more efficient than both, as the CTL\* algorithm is more complex, and LTL  
 523 algorithms tend to have exponential complexity. Additionally, composing CTL properties is  
 524 somewhat more challenging than composing LTL properties, which can also be expressed using  
 525 CTL\*. While CTL\* is more expressive than CTL, it still has certain limitations compared to LTL,  
 526 particularly concerning the structure of temporal formulas and the types of properties that can  
 527 be expressed. Therefore, when choosing between LTL, CTL, or CTL\* for security property  
 528 specification, one must consider efficiency, comprehensibility, and expressiveness based on the  
 529 size of the ACPs and the complexities involved (i.e., static versus dynamic and single versus  
 530 combinations). The relationships among LTL, CTL, and CTL\* are illustrated in Fig. 12 along with  
 531 example formulas.



532

533

Fig. 12. Relationships among LTL, CTL, and  $CTL^*$

### 534 3.2. Security Properties

535 From the perspective of ACP property verification, LTL is more suitable for expressing security  
536 properties that can be evaluated over a single linear path, such as “eventually, a permission  
537 decision is made,” or “always, no invalid access occurs.” For combined ACPs, LTL can effectively  
538 verify security properties for intersection and concatenation types of nonconcurrent ACPs as  
539 well as for independent and shared action types of concurrent ACPs because the transition  
540 model of these combinations does not branch out into separate paths unless it involves  
541 dynamic COI ACPs. However, LTL properties may not be sufficient for verifying combined  
542 automata that loop in a sequence without passing through others, which can be addressed by  
543 CTL (or  $CTL^*$ ) (see Sec. 3.1.3). Thus, CTL (or  $CTL^*$ ) can be applied to the union of nonconcurrent  
544 ACPs and shared variables in concurrent types of ACP combinations, as both will branch out  
545 into different paths, regardless of whether the ACPs are static or dynamic. If a security check  
546 involves multiple combined or mixed ACPs, even if LTL is sufficient, it is reasonable to first  
547 consider CTL because its expressive capability is superior to LTL while still being more efficient  
548 than  $CTL^*$  in terms of algorithm complexity.

549 Security properties are formally specified to identify faults in ACP models that may lead to  
550 privilege leaks or block authorized access. The two main categories of property checks — safety  
551 and liveness — are applied to access control security property assessments using LTL and CTL to  
552 detect faults in the automata of the ACP’s transition models.

#### 553 3.2.1. Safety

554 In model checking, safety refers to the assurance that something undesirable never occurs. This  
555 is a fundamental property of an ACP that ensures the absence of safety threats, including  
556 privilege leakage, privilege conflicts, and privilege escalation to unauthorized or unintended  
557 principals. Safety can be specified using LTL or CTL languages, which can generally be proven for  
558 ACP transition models that describe the safety requirements of any configuration [IR7874].  
559 Formally, a safety property  $p$  in LTL or CTL is said to satisfy an ACP transition automaton  $A$  if  
560 there is no violation of the rules defined by the logic in  $p$ . It is assumed that  $A$  will eventually

561 reach an accepted permission state after taking actions that comply with input user access  
562 requests. If certain properties cannot be expressed in LTL or CTL, they cannot be verified, as the  
563 verification algorithms are limited to handling regular expressions (i.e., invariants) that are  
564 defined by the associated function *BadPrefixed Set* ( $\neg p$ ) [SF].

565 An example of safety properties for ACP with random access rules is to ensure that all access  
566 requests that comply with specified constraints are granted, while all non-compliant requests  
567 are denied. The system state for access authorization is initialized as the deny state and  
568 transitions to the grant state for any access request that meets the constraints outlined in the  
569 corresponding rule (i.e., *constraint 1 . . . AND constraint n*). The system remains in the *deny*  
570 state for any requests that do not comply. The properties of the static constraints can be  
571 verified using the following CTL properties:

572  $AG (constraint\ 1 \ \& \ constraint\ 2 \ \& \ \dots \ constraint\ n) \rightarrow AX (access\ state = 1)$   
573  $AG (constraint\ a \ \& \ constraint\ b \ \& \ \dots \ constraint\ m) \rightarrow AX (access\ state = 1) \dots\dots$   
574  $AG ! ((constraint\ 1 \ \& \ \dots \ constraint\ n) / (constraint\ a \ \& \ \dots \ constraint\ m) / \dots) \rightarrow$   
575  $AX (access\ state = 0)$

576 Specifications of the form “AG ( $p$ )  $\rightarrow$  AX ( $q$ )” indicate that for all paths (the “A” in “AG”) and for  
577 all states globally (the “G”), if  $p$  holds, then (“ $\rightarrow$ ”) in the next state (the “X” in “AX”),  $q$  will hold  
578 for all paths [JO].

579 SOD is another safety property that is more dynamic than others. It refers to the principle that  
580 no user should be granted enough privileges to independently misuse the system. For example,  
581 the person who authorizes paychecks should not also be the one who can prepare them. SOD  
582 can be enforced either statically (i.e., by defining conflicting roles that cannot be executed by  
583 the same user) or dynamically (i.e., by enforcing control at the time of access). An example of a  
584 CTL property is  $G(\neg critical1 \vee \neg critical2)$ , which specifies that processes cannot be in the *critical*  
585 *section* simultaneously in a semaphore scheme for *processes 1* and *2*, where *critical1* represents  
586 that *process 1* is in the *critical section*, and *critical2* represents that *process 2* is in the *critical*  
587 *section* [SP192]

### 588 3.2.2. Liveness

589 In model checking, liveness refers to the guarantee that something good eventually happens,  
590 ensuring that a transition model does not encounter a deadlock (i.e., where the system waits  
591 indefinitely for an event) or a livelock (i.e., where the model repeatedly executes the same  
592 operations without progress). An example of a livelock is the Dining Philosophers problem 26 in  
593 which philosophers could endlessly alternate between thinking and trying to eat without ever  
594 succeeding, often due to issues with scheduler fairness in concurrency.

595 Threats to liveness in an ACP include privilege blocking and cyclic inheritance (e.g., a Workflow  
596 dynamic ACP could cause a deadlock if the work process involves cyclic dependencies). The  
597 liveness check for an ACP determines whether every access control request will eventually

598 receive a meaningful decision (e.g., grant, denial, or other action). Temporal and quantifier  
599 operators used in LTL or CTL for liveness verification include:

- 600 •  $G p$ : Always  $p$
- 601 •  $F p$ : Sometimes  $p$
- 602 •  $G F p$ : Infinitely often  $p$
- 603 •  $A F G p$ : Infinitely often  $p$  for all paths

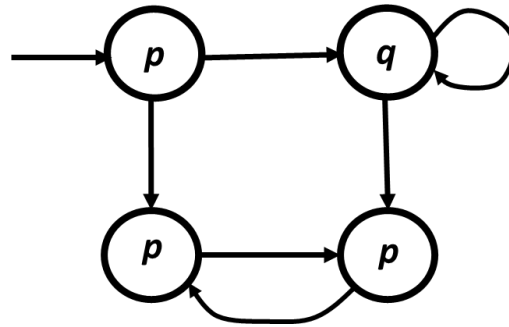
604 Here,  $p$  represents an accepting access control decision (e.g., grant, denial, or another  
605 meaningful action). For example, the CTL property  $GF \textit{critical1} \wedge GF \textit{critical2}$  specifies that each  
606 process visits the *critical* section infinitely often in a semaphore scheme for *processes 1* and *2*,  
607 where *critical1* indicates that *process 1* is in the *critical* section, and *critical2* indicates that  
608 *process 2* is in the *critical* section.

609 **4. Verification Process**

610 This section introduces the general method and NuSMV tool for checking ACP transition  
 611 models.

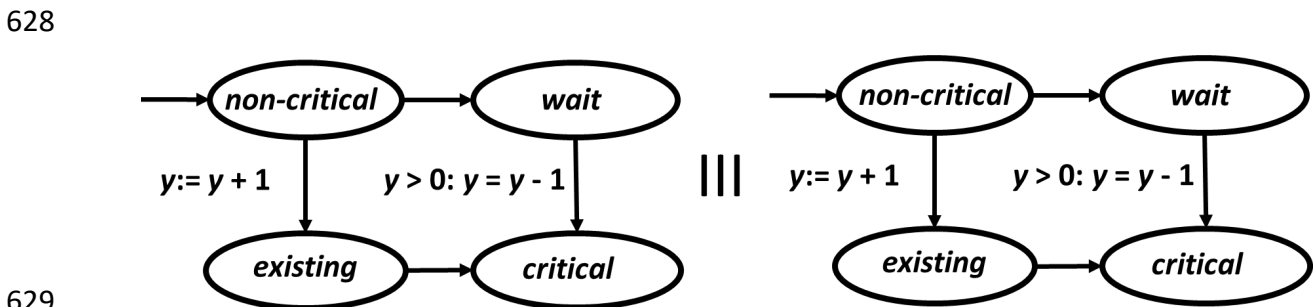
612 **4.1. General Method**

613 A property is an invariant that must hold true throughout the execution of a system, such as  
 614 “the property  $p$  is always true.” Since ACPs can be translated into transition models (see Sec. 2),  
 615 verifying a security property involves checking whether it can be satisfied by the automaton of  
 616 an ACP’s transition model (i.e., all traces in the transition model satisfy the property  $p$ ).  
 617 Formally, a transition model  $TS$  satisfies a security property  $p$  if  $Trace(TS) \subseteq p$ , where  $Trace(TS)$   
 618 represents all possible executions of the transition model’s state change path. For example, Fig.  
 619 13 shows a transition model that satisfies the CTL property  $EG\ p$  but not  $AF\ q$ .



620  
 621 **Fig. 13. Example of the ACP transition model that satisfies  $EG\ p$  but not  $AF\ q$**

622 Consider a mutual exclusion access control system with atomic propositions  $AP = \{s1, s2, s3,$   
 623  $s4\}$ , where  $s1$  represents “process 1 is in the *critical state*,”  $s2$  represents “process 1 is in the  
 624 *wait state*,”  $s3$  represents “process 2 is in the *critical state*,” and  $s4$  represents “process 2 is in  
 625 the *wait state*.” The transition model of the mutual exclusion ACP, as shown in Fig. 14, satisfies  
 626 the CTL property  $AG\ \neg (s1 \wedge s3)$ , which means that in all paths of the transition model,  $s1$  and  
 627  $s3$  will never occur simultaneously [BK].



629  
 630 **Fig. 14. A mutual exclusion access system**

631 Checking the safety of an automaton in an ACP transition model involves verifying that no  
 632 forbidden or error states (indicative of access faults) can be reached from the initial state under  
 633 any sequence of transitions. The first step in the verification process is to analyze the

634 automaton’s structure to identify which states are considered faults according to the ACP’s  
635 security requirements. Next, code is implemented using a graph traversal algorithm (e.g.,  
636 depth-first search [DFS] or breadth-first search [BFS]) that is applied to the automaton’s graph  
637 representation, where access states are nodes and access transitions are edges.

638 Checking an automaton’s liveness in an access control system involves verifying that it is  
639 possible to eventually reach an access request decision state from every reachable state. The  
640 first step in the verification process is to analyze the automaton’s structure to identify which  
641 states are access request decision (accepting) states. Then, either implement code as described  
642 for the safety check above to determine whether there is a path from each state to an  
643 accepting state, or check whether each state in the original automaton is reachable in the  
644 reverse automaton starting from any accepting state.

645 To support the implementation of verification algorithms, tools such as the NetworkX (Python)  
646 library [NX] for graph representations and traversal algorithms and AutomataLib (Java) [AJ] or  
647 the Automata package (Python) [AP] for handling automata-related operations can be used.  
648 Additionally, automata can be formally described and verified using tools such as NuSMV [NU]  
649 or SPIN [SP].

## 650 **4.2. NuSMV Tool**

651 NuSMV [NU] is a symbolic model checker that was developed by the Formal Methods and Tools  
652 group at the University of Trento and Cadence Berkeley Labs. It is used to formally verify finite-  
653 state systems and supports the verification of systems modeled in hardware description  
654 languages, software systems, protocols, and safety-critical systems. Widely used in both  
655 academia and industry, NuSMV offers robust capabilities for various verification needs. It allows  
656 users to define systems in a modular fashion using the SMV language, which is based on the  
657 concept of transition model verification. NuSMV checks whether the model satisfies the  
658 specified properties using CTL path quantifiers or LTL temporal logic formulas. If a property is  
659 violated, it provides a counterexample to help identify the issue [SP192].

660 NIST’s Access Control Policy Tool (ACPT) [AC][HX] utilizes NuSMV to provide access control  
661 security requirements verification for both static and dynamic ACPs in various combinations.  
662 ACPT helps eliminate the possibility of creating faulty access control models that could either  
663 leak information or prohibit legitimate information sharing. Similarly, NuSMV is used by other  
664 ACP verification tools, such as MOHAWK [MO][SP192].

## 665 **4.3. Comparison With Other Model-Checking Methods**

666 Other model-checking methods applied to ACP security property verification have their own  
667 trade-offs when compared to traditional model checking. For instance, Margrave [CL] is a  
668 software tool suite that was designed to verify safety requirements against ACPs written in  
669 XACML [XA]. Margrave represents XACML ACPs as multi-terminal binary decision diagrams  
670 (MTBDDs) and allows users to specify various forms of safety requirements in the Scheme  
671 programming language. Margrave’s API can verify these safety properties, and if there are any  
672 counterexamples that violate the properties, they are produced. The chief innovation of

673 Margrave’s approach lies in its use of full first-order predicate logic, which quantifies individuals  
674 in a domain and reasons about their properties and relationships using quantifiers like “for all”  
675 ( $\forall$ ) and “there exists” ( $\exists$ ).

676 Margrave supports query-based verification and provides query-based views by computing  
677 exhaustive sets of scenarios that yield different results. It offers the benefits of static  
678 verification without requiring authors to write formal properties. Its strength comes from  
679 selecting an appropriate policy model in first-order logic and embracing both scenario-finding  
680 and multi-level policy reasoning.

681 The Z language, commonly known as Z notation [ZL], is based on axiomatic set theory and first-  
682 order logic, making it suitable for describing and modeling ACPs [HU]. In Z notation, creating an  
683 AC model involves using set theory to provide a robust foundation that allows for specifications  
684 to be structured and modularized. Schemas are used to encapsulate access control state  
685 variables and their invariants as well as operations that modify the state. This approach  
686 supports syntax and type checking, schema expansion, precondition calculation, domain  
687 checking, and general theorem proving for model verification. Many proof obligations are easily  
688 proven, and even in more challenging cases, generating the proof obligation significantly aids in  
689 determining whether a property specification in the AC model is meaningful.

690 In terms of specifying security properties on ACP transition automata, the LTL and path  
691 quantifier properties of CTL are not classified as first-order logic properties compared to the  
692 other major model-checking methods. However, they can express many properties that first-  
693 order logic cannot and can be applied to both static and dynamic ACP models. Additionally,  
694 when applied to different ACP models by combining their transition models, property  
695 specification that uses LTL and CTL provides well-defined rules for operations that other  
696 methods lack.



697 **5. Conclusion**

698 This document explains how to apply model-checking techniques to verify security properties in  
699 ACPs. It begins with a brief introduction to the fundamentals of model checking and  
700 demonstrates how ACPs are converted into automata through their transition models. The  
701 document then discusses property specifications in terms of LTL and CTL with comparisons  
702 between the two. This is followed by an examination of access control security properties using  
703 both logics. Finally, the verification process and available tools are described and compared.

704

705 **References**

- 706 [AC] National Institute of Standards and Technology (2024) Access Control Policy Testing:  
707 Beta Release of Access Control Policy Tool. Available at  
708 <https://csrc.nist.gov/projects/access-control-policy-tool/beta-release-of-access->  
709 [control-policy-tool](https://csrc.nist.gov/projects/access-control-policy-tool/beta-release-of-access-control-policy-tool)
- 710 [AJ] TU Dortmund University (2024) AutomataLib. Available at  
711 <https://github.com/LearnLib/automatalib>
- 712 [AP] Python Software Foundation (2024) automata-lib 8.4.0. Available at  
713 <https://pypi.org/project/automata-lib/>
- 714 [BK] Baier C, Katoen JP (2008) Principles of Model Checking (MIT Press, Cambridge, MA).  
715 Available at [https://mitpress.mit.edu/9780262026499/principles-of-model-](https://mitpress.mit.edu/9780262026499/principles-of-model-checking/)  
716 [checking/](https://mitpress.mit.edu/9780262026499/principles-of-model-checking/)
- 717 [CL] Clarke EM, Fujita M, McGeer PC, McMillan K, Yang JC, Zhao X (1993) Multi-terminal  
718 binary decision diagrams: An efficient data structure for matrix representation, *IWLS*  
719 *'93: International Workshop on Logic Synthesis (IWLS '93)* (Tahoe City, CA). Available  
720 at  
721 [http://www.cs.cmu.edu/afs/cs/Web/People/emc/papers/Technical%20Reports/Mul-](http://www.cs.cmu.edu/afs/cs/Web/People/emc/papers/Technical%20Reports/MultiTerminal%20Binary%20Decision%20Diagrams%20An%20Efficient%20Data%20Structure%20for%20Matrix%20Representation.pdf)  
722 [tiTerminal%20Binary%20Decision%20Diagrams%20An%20Efficient%20Data%20Structu-](http://www.cs.cmu.edu/afs/cs/Web/People/emc/papers/Technical%20Reports/MultiTerminal%20Binary%20Decision%20Diagrams%20An%20Efficient%20Data%20Structure%20for%20Matrix%20Representation.pdf)  
723 [re%20for%20Matrix%20Representation.pdf](http://www.cs.cmu.edu/afs/cs/Web/People/emc/papers/Technical%20Reports/MultiTerminal%20Binary%20Decision%20Diagrams%20An%20Efficient%20Data%20Structure%20for%20Matrix%20Representation.pdf)
- 724 [HR] Huth M, Ryan M (2004) Logic in Computer Sciences – Modelling and Reasoning  
725 about Systems, *Cambridge Section 3.6.1*. Available at  
726 [https://downloads.regulations.gov/PTO-P-2021-0032-0138/attachment\\_2.pdf](https://downloads.regulations.gov/PTO-P-2021-0032-0138/attachment_2.pdf)
- 727 [HU] Hu VC (2002) Dissertation: The Policy Machine For Universal Access Control,  
728 *Computer Science Department, University of Idaho*. Available at  
729 <https://dl.acm.org/doi/10.5555/936065>
- 730 [HX] Hwang J, Xie T, Hu V, Altunay M (2010) ACPT: A Tool for Modeling and Verifying  
731 Access Control Policies. *2010 IEEE International Symposium on Policies for*  
732 *Distributed Systems and Networks (POLICY 2010)* (IEEE, Fairfax, VA), pp. 40-43.  
733 <https://doi.org/10.1109/POLICY.2010.22>
- 734 [IR7316] Hu VC, Ferraiolo DF, Kuhn DR (2006) Assessment of Access Control Systems.  
735 (National Institute of Standards and Technology, Gaithersburg, MD), NIST  
736 Interagency or Internal Report (IR) 7316. <https://doi.org/10.6028/NIST.IR.7316>
- 737 [IR7874] Hu VC, Scarfone K (2012) Guidelines for Access Control System Evaluation Metrics  
738 (National Institute of Standards and Technology, Gaithersburg, MD), NIST  
739 Interagency or Internal Report (IR) 7874. <https://doi.org/10.6028/NIST.IR.7874>
- 740 [JO] Hu VC, Kuhn DR, Xie T, Hwang J (2011) Model Checking for Verification of  
741 Mandatory Access Control Models and Properties. *International Journal of Software*  
742 *Engineering and Knowledge Engineering* 21(1):103–127.  
743 <https://doi.org/10.1142/S021819401100513X>
- 744 [MO] Jayaraman K, Ganesh V, Tripunitara M, Rinard M, Chapin S (2011) Mohawk:  
745 Automatic Verification of Access-Control Policies. *Proceedings of the 18<sup>th</sup> ACM*  
746 *Conference on Computer and Communications Security (CCS '11)* (ACM, Chicago, IL),  
747 pp. 163-174. <https://dl.acm.org/doi/10.1145/2046707.2046727>

- 748 [NU] Fondazione Bruno Kessler (2023) NuSMV: a new symbolic model checker. Available  
749 at <https://nusmv.fbk.eu/>
- 750 [NX] NetworkX developers (2024) NetworkX: Network Analysis in Python. Available at  
751 <https://networkx.org/>
- 752 [SF] NPTEL (2015) Safety properties described by automata. Available at  
753 <https://www.youtube.com/watch?v=DzbqwlUw2-g>
- 754 [SP] Spin developers (2024) Verifying Multi-threaded Software with Spin. Available at  
755 <https://spinroot.com/spin/whatispin.html>
- 756 [SP162] Hu VC, Ferraiolo DF, Kuhn DR, Schnitzer A, Sandlin K, Miller R, Scarfone KA (2014)  
757 Guide to Attribute Based Access Control (ABAC) Definition and Considerations.  
758 (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special  
759 Publication (SP) 800-162, Includes updates as of August 02, 2019.  
760 <https://doi.org/10.6028/NIST.SP.800-162>
- 761 [SP192] Hu VC, Kuhn DR, Yaga D (2017) Verification and Test Methods for Access Control  
762 Policies/Models. (National Institute of Standards and Technology, Gaithersburg,  
763 MD), NIST Special Publication (SP) 800-192. <https://doi.org/10.6028/NIST.SP.800-192>
- 764
- 765 [WP] Wikipedia (2024) Dining philosophers problem. Available at  
766 [https://en.wikipedia.org/wiki/Dining\\_philosophers\\_problem](https://en.wikipedia.org/wiki/Dining_philosophers_problem)
- 767 [XA] OASIS Open (2023) XACML resources. Available at <http://docs.oasis-open.org/xacml/>
- 768 [YC] CTL\* (2016) *NPTEL National Program on Technology Enhanced learning*. Available at  
769 <https://www.youtube.com/watch?v=2E5Q3CZ7g4&t=56s>
- 770 [YC2] CTL (2016) *National Program on Technology Enhanced learning*. Available at  
771 <https://www.youtube.com/watch?v=Blh060Hgbm8>
- 772 [ZL] Potter B, Sinclair J, Till D (1996) An Introduction to Formal Specification and Z  
773 (Prentice Hall PTR, Upper Saddle River, NJ), 2nd Ed. Available at  
774 <https://dl.acm.org/doi/10.5555/547639>