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

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

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

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

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.

Abstract

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

Keywords

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

Reports on Computer Systems Technology

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

Call for Patent Claims

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

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

a) assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or

b) assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:

   i. under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or

   ii. without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

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

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

Such statements should be addressed to ir8539-comments@nist.gov

Table of Contents

List of Tables

List of Figures

Acknowledgments

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

Executive Summary

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

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

1. Introduction

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

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

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

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

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

2. Formal Models and ACPs

This section explains the application of formal models to ACPs.

2.1. Model Fundamentals

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

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

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

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

2.2. ACP Automata

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

2.2.1. Static ACPs

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

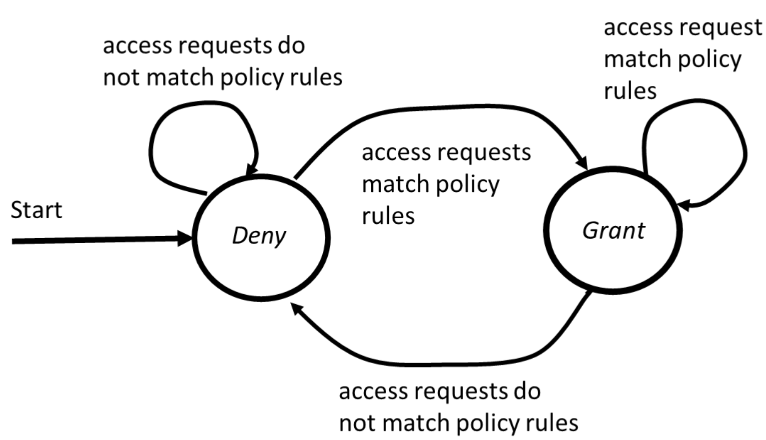

Fig. 1. Example automaton of a random rules ACP

In the automaton, the access authorization state is initialized as the deny state and transitions to the grant state for any access request that complies with the rule's constraints. Otherwise, it remains in the deny state. Even though environmental condition variables (e.g., time, location) may change through monitoring, they do not affect state transitions from the perspective of the automaton.

2.2.2. Dynamic ACPs

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

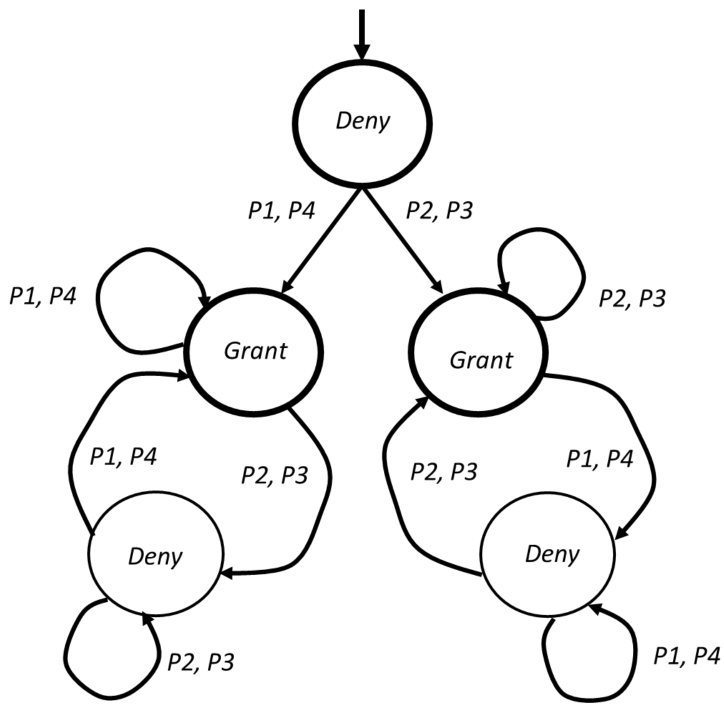

Fig. 2. Example automaton of a Chinese Wall ACP

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

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

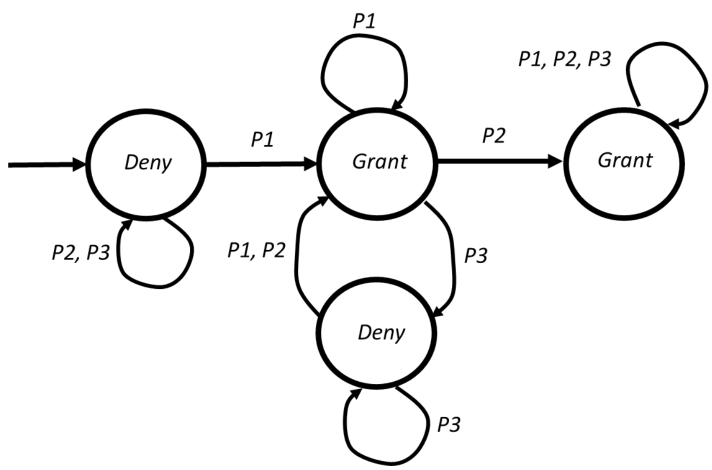

Fig. 3. Example automaton of a Workflow ACP

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

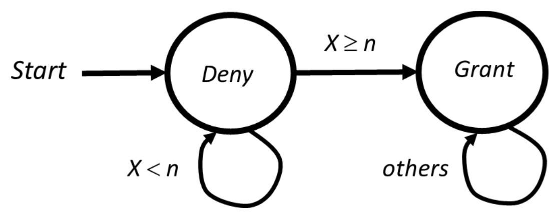

Fig. 4. Example automaton of an N-person Control ACP

2.3. ACP combinations

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

In general, ACP combinations can be divided into two categories: nonconcurrent and concurrent combinations. A nonconcurrent combination does not require synchronization between the combined ACPs, while a concurrent combination does need to account for the synchronization of shared states or variables between the ACPs. To ensure that the combined ACP functions correctly, it is essential to formally detect any inconsistencies or incompleteness, such as scenarios in which an access request is both accepted and denied or where the request is neither accepted nor denied according to the combined automata [SP192]. Each type of combination has distinct characteristics.

2.3.1. Nonconcurrent ACP Combinations

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

2.3.1.1. Intersection

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

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

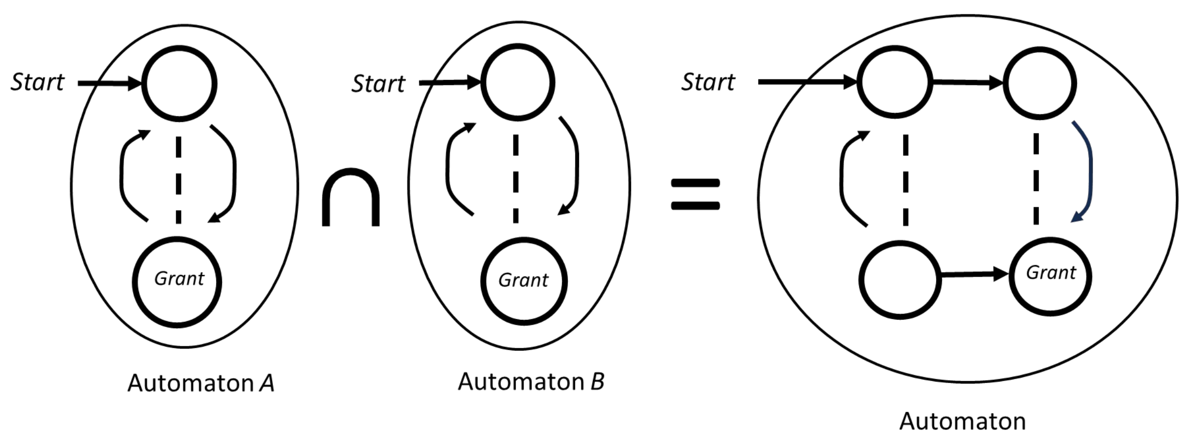

Fig. 5. Intersection concept using an example of two automata

2.3.1.2. Union

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

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

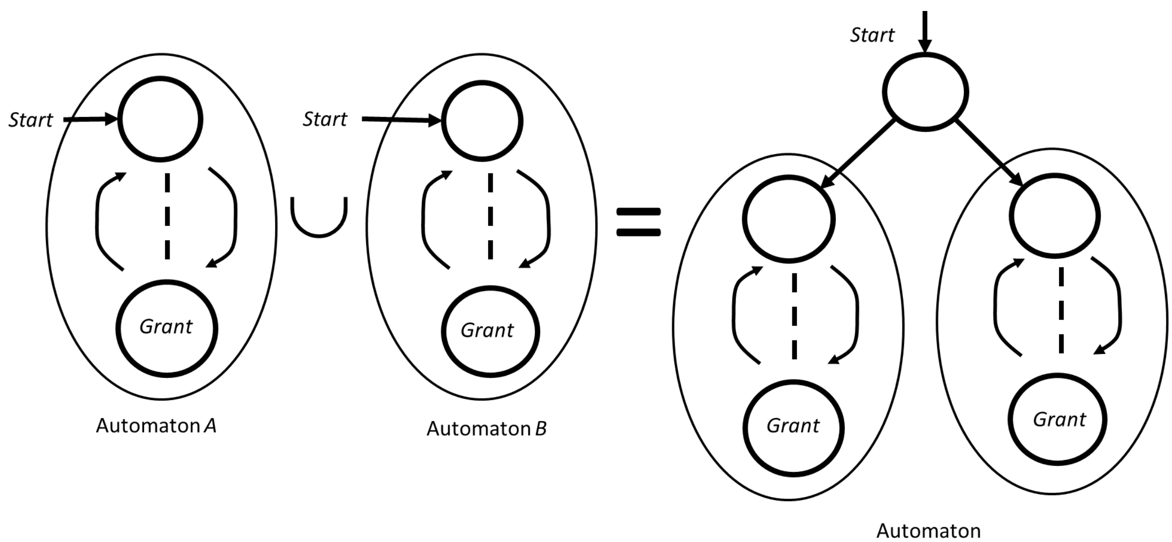

Fig. 6. Union concept using an example of two automata

2.3.1.3. Concatenation

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

Let A_i represent the automaton of ACP_i, which corresponds to work unit W_i. The concatenation of automata A_1, A_2,…., A_n involves connecting these automata end to end such that the output of the first automaton serves as the input to the next. Formally, $A_1 + A_2 .... + A_n$ denotes the automaton that results from linking the accepting states of A_1 to the initial state of A_2, the accepting states of A_2 to the initial state of A_3, and so on. This process effectively creates a new automaton that recognizes strings accepted first by A_1, which then pass through the subsequent concatenated automata until reaching A_n. Figure 7 illustrates the concept of concatenation with an example of two automata: $A + B$.

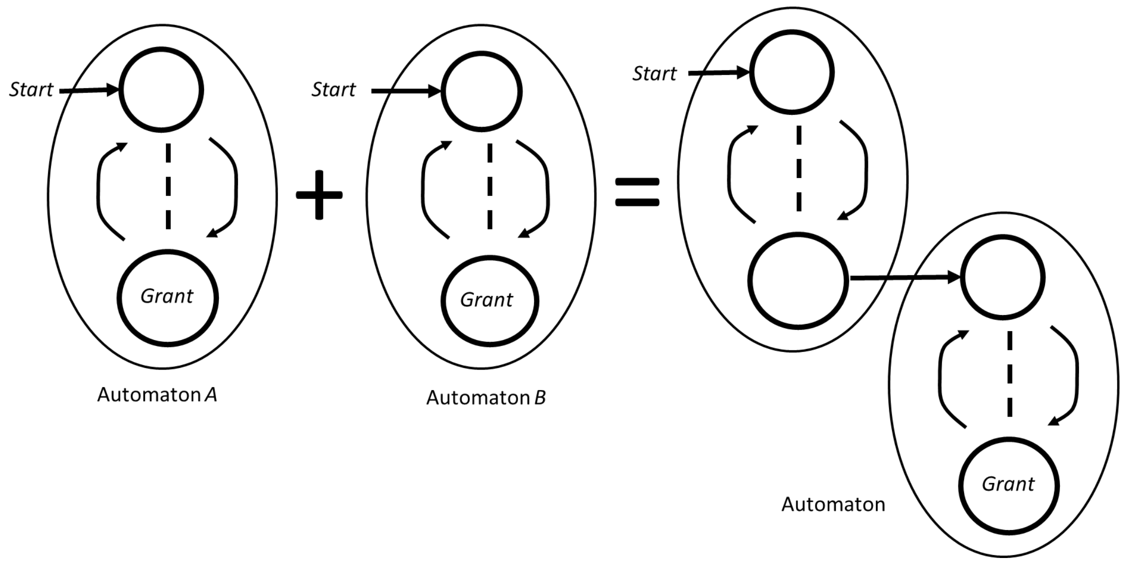

Fig. 7. Concatenation concept using an example of two automata

2.3.2. Concurrent ACP Combinations

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

2.3.2.1. Independent

Some situation-awarded ACPs rely on synchronized states to determine access permissions (e.g., air traffic control systems monitor multiple runway situations to manage access to runways). Such systems can employ independent concurrent automata for safety checks, similar to a traffic light system that only permits specific light combinations. The interleaving of independent systems operates in a way that allows their states to change dynamically and interleave with one another, meaning that the authorization processes run independently and disregards the order in which they are executed. This type of combination is a variant of the nonconcurrent intersection combinations (see Sec. 2.3.1). Each ACP has its own set of environment variables so that instead of sharing variables or actions, they share system states. Formally, A_1 III A_2…. III A_n, where A_i represents the automaton of ACP_i for concurrent access control system S_i's ACP, and the symbol III is the interleaving operator. Figure 8 illustrates the concept of independent concurrency with an example of two automata: *A* III *B* [BK].

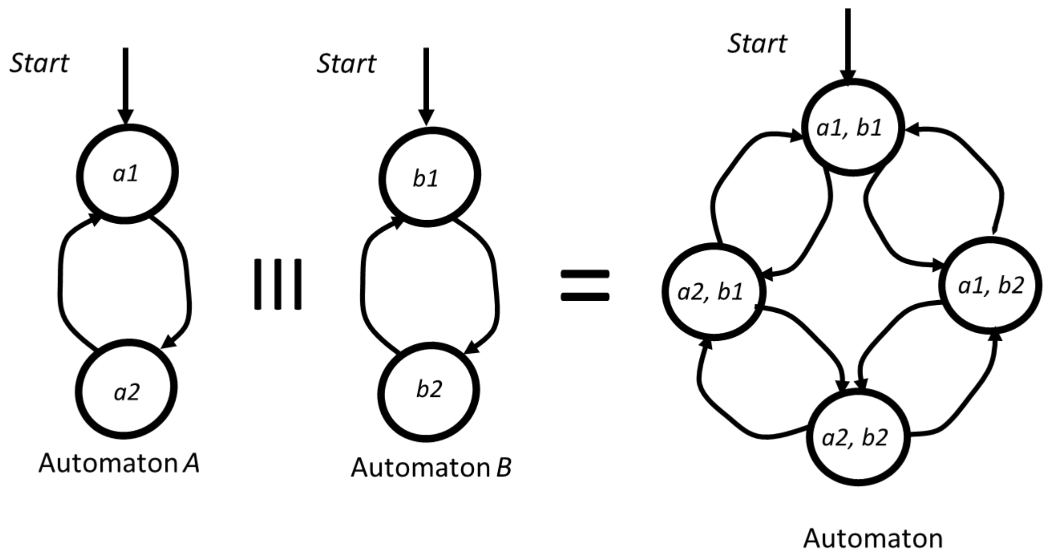

Fig. 8. Example of a combination of two interleaving automata

2.3.2.2. Shared Variables

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

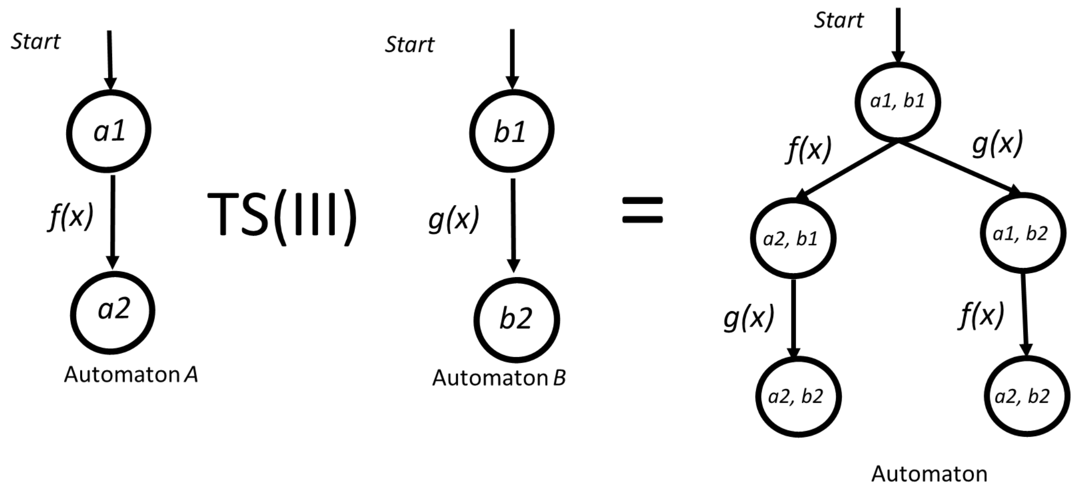

Fig. 9. Shared variables concept with an example of two automata

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

```
{ VARIABLES
count, access_limit : INTEGER;
request_1 : process_request (count);
request_2 : process_request (count);
        …….
request_n: process_request (count);
/*max number of user requests allowed by the system*/
access_limit := k; /*max number of concurrent access*/
count := 0; act {rd, wrt}; object {obj};
process_request (access_limit) {
        VARIABLES
        permission : {start, grant, deny};
        state : {idle, entering, critical, exiting};
        INITIAL_STATE (permission) := start;
        INITIAL_STATE (state) := idle;
        NEXT_STATE (state) := CASE {
                state == idle : {idle, entering};
                state == entering & ! (count > access_limit): critical;
                state == critical : {critical, exiting};
```

```
            state == exiting : idle;
            OTHERWISE: state};
    NEXT_STATE (count) := CASE {
            state == entering : count + 1;
            state == exiting : count -1;
            OTHERWISE: DO_NOTHING };
    NEXT_STATE (permission) := CASE {
            (state == entering)& (act == rd) & (object == obj): grant;
            OTHERWISE: deny;}
    }
}
```

2.3.2.3. Shared actions

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

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

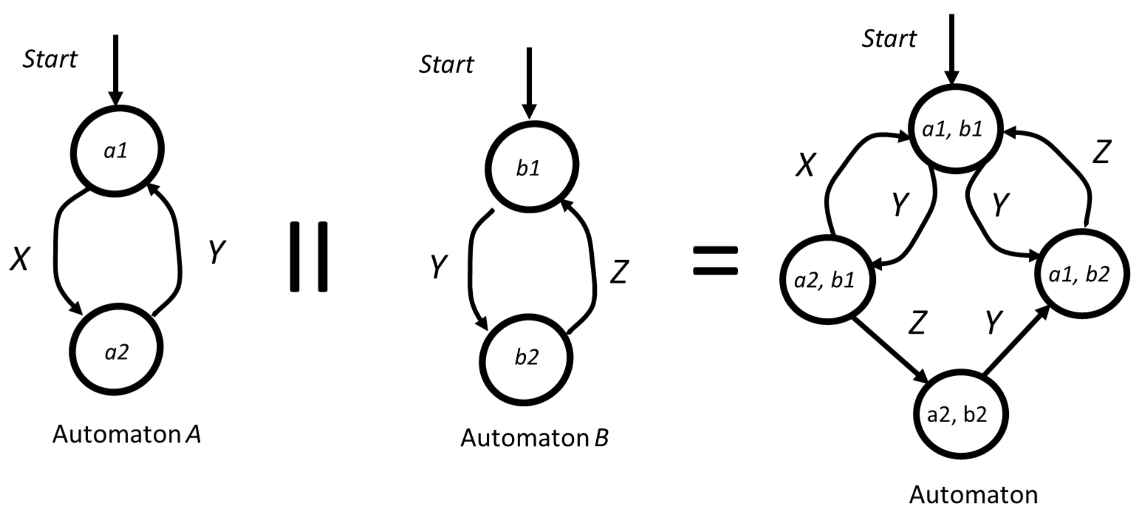

Fig. 10. Shared actions concept with an example of two automata

Concurrent automata are constructed from multiple transition models of ACPs. An accepting state (e.g., grant or deny) of the combined automata must be one of the combinations of the individual accepting states from all of the automata. In the worst-case scenario, for *n* ACPs, the maximum number of states of the combined automata is $O(2^n)$.

3. Properties

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

3.1. Property Specifications

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

1. Path quantifiers or temporal operators, such as U, G, F, and X
2. Finite automata
3. Regular expressions, including ω-regular expressions

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

3.1.1. Linear Temporal Logic (LTL)

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

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

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

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

3.1.2. Computation Tree Logic (CTL)

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

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

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

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

- AX (For all next): AX *p* means *p* holds in all next states.
- EX (Exists next): EX *p* means that there exists a next state where *p* holds.
- AF (For all future): AF *p* means *p* will eventually hold on all paths.
- EF (Exists future): EF *p* means there exists a path where *p* will eventually hold.
- AG (For all globally): AG *p* means *p* holds globally on all paths.
- EG (Exists globally): EG *p* means there exists a path where *p* holds globally.
- A (*p* U *q*): Means *p* holds until *q* holds on all paths.
- E (*p* U *q*): Means there exists a path where *p* holds until *q* holds.

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

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

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

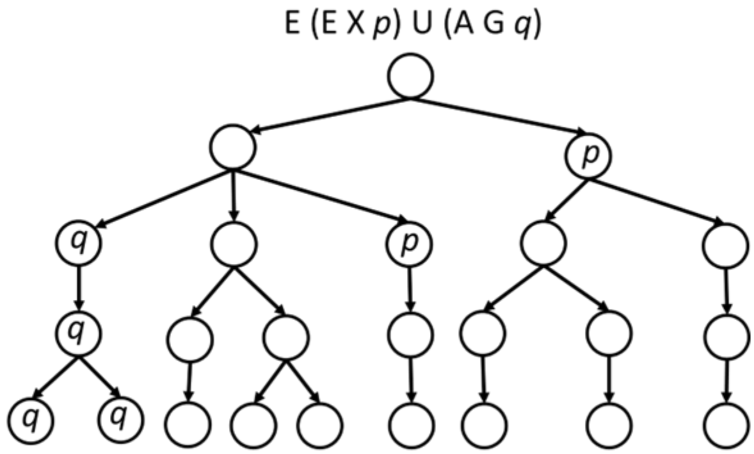

Fig. 11. Example of E(EX *p*) U (AG *q*) in CTL

3.1.3. Computation Tree Logic Star (CTL)

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

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

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

- CTL grammar
    - State formulae: ϕ := true|p_i | $\phi_1 \wedge \phi_2$ | $\neg\, \phi_1$ | E α | A α
    - Path formulae: α := X ϕ_1 | ϕ_1 U ϕ_2 | F ϕ_1 | G α_1
- CTL* grammar

- State formulae: ϕ := true | p_i | $\phi_1 \wedge \phi_2$ | $\neg \phi_1$ | E α | A α
- Path formulae: α := ϕ | $\alpha_1 \wedge \alpha_2$ | $\neg \alpha_1$ | X α_1 | α_1 U α_2 | F α_1 | G α_1, where ϕ, ϕ_1 and ϕ_2 are state formulae, and α, α_1 and α_2 are path formulae

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

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 ∧ A G q	A (F p ∧ G q)
A (p U (E G q))	A (p U (G q))

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

3.1.4. LTL vs. CTL (and CTL*)

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

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

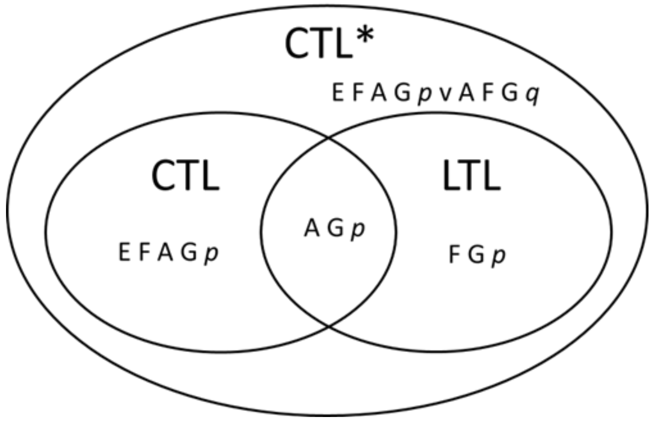

Fig. 12. Relationships among LTL, CTL, and CTL*

3.2. Security Properties

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

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

3.2.1. Safety

In model checking, safety refers to the assurance that something undesirable never occurs. This is a fundamental property of an ACP that ensures the absence of safety threats, including privilege leakage, privilege conflicts, and privilege escalation to unauthorized or unintended principals. Safety can be specified using LTL or CTL languages, which can generally be proven for ACP transition models that describe the safety requirements of any configuration [IR7874]. Formally, a safety property *p* in LTL or CTL is said to satisfy an ACP transition automaton *A* if there is no violation of the rules defined by the logic in *p*. It is assumed that *A* will eventually reach an accepted permission state after taking actions that comply with input user access requests. If certain properties cannot be expressed in LTL or CTL, they cannot be verified, as the verification algorithms are limited to handling regular expressions (i.e., invariants) that are defined by the associated function *BadPrefixed Set* ($\neg p$) [SF].

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

$$AG\ (constraint\ 1\ \&\ constraint\ 2\ \&\ \ldots.\ constraint\ n)\ \rightarrow AX\ (access\ state = 1)$$

$$AG\ (constraint\ a\ \&\ constraint\ b\ \&\ \ldots.\ constraint\ m) \rightarrow AX\ (access\ state = 1)\ \ldots\ldots$$

$$AG\ !\ ((constraint\ 1\ \&\ \ldots.constraint\ n)\ /\ (constraint\ a\ \&\ \ldots.\ constraint\ m)\ /\ldots)\ \rightarrow$$
$$AX\ (access\ state = 0)$$

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

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

3.2.2. Liveness

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

Threats to liveness in an ACP include privilege blocking and cyclic inheritance (e.g., a Workflow dynamic ACP could cause a deadlock if the work process involves cyclic dependencies). The liveness check for an ACP determines whether every access control request will eventually receive a meaningful decision (e.g., grant, denial, or other action). Temporal and quantifier operators used in LTL or CTL for liveness verification include:

- G *p*: Always *p*
- F *p*: Sometimes *p*
- G F *p*: Infinitely often *p*
- A F G *p*: Infinitely often *p* for all paths

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

4. Verification Process

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

4.1. General Method

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

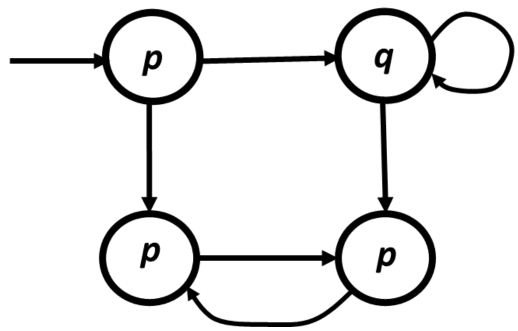

Fig. 13. Example of the ACP transition model that satisfies EG *p* but not AF *q*

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

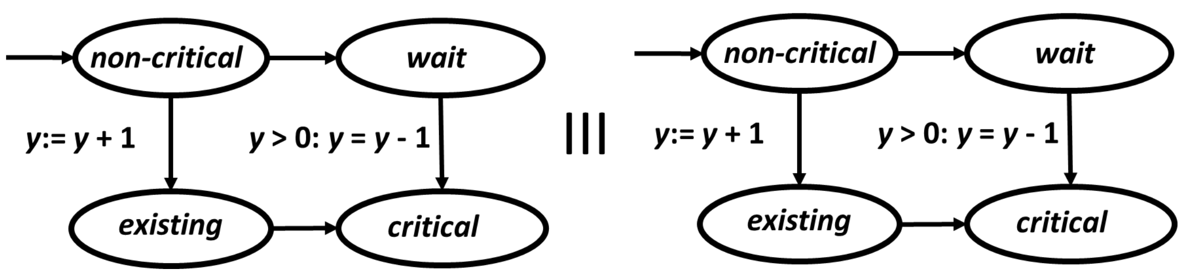

Fig. 14. A mutual exclusion access system

Checking the safety of an automaton in an ACP transition model involves verifying that no forbidden or error states (indicative of access faults) can be reached from the initial state under any sequence of transitions. The first step in the verification process is to analyze the automaton's structure to identify which states are considered faults according to the ACP's security requirements. Next, code is implemented using a graph traversal algorithm (e.g., depth-first search [DFS] or breadth-first search [BFS]) that is applied to the automaton's graph representation, where access states are nodes and access transitions are edges.

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

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

4.2. NuSMV Tool

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

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

4.3. Comparison With Other Model-Checking Methods

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

Margrave's approach lies in its use of full first-order predicate logic, which quantifies individuals in a domain and reasons about their properties and relationships using quantifiers like "for all" (∀) and "there exists" (∃).

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

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

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

5. Conclusion

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

References

[AC] National Institute of Standards and Technology (2024) Access Control Policy Testing: Beta Release of Access Control Policy Tool. Available at https://csrc.nist.gov/projects/access-control-policy-tool/beta-release-of-access-control-policy-tool

[AJ] TU Dortmund University (2024) AutomataLib. Available at https://github.com/LearnLib/automatalib

[AP] Python Software Foundation (2024) automata-lib 8.4.0. Available at https://pypi.org/project/automata-lib/

[BK] Baier C, Katoen JP (2008) Principles of Model Checking (MIT Press, Cambridge, MA). Available at https://mitpress.mit.edu/9780262026499/principles-of-model-checking/

[CL] Clarke EM, Fujita M, McGeer PC, McMillan K, Yang JC, Zhao X (1993) Multi-terminal binary decision diagrams: An efficient data structure for matrix representation, *IWLS '93: International Workshop on Logic Synthesis (IWLS '93)* (Tahoe City, CA). Available at http://www.cs.cmu.edu/afs/cs/Web/People/emc/papers/Technical%20Reports/MultiTerminal%20Binary%20Decision%20Diagrams_An%20Efficient%20Data%20Structure%20for%20Matrix%20Representation.pdf

[HR] Huth M, Ryan M (2004) Logic in Computer Sciences – Modelling and Reasoning about Systems, *Cambridge Section 3.6.1*. Available at https://downloads.regulations.gov/PTO-P-2021-0032-0138/attachment_2.pdf

[HU] Hu VC (2002) Dissertation: The Policy Machine For Universal Access Control, *Computer Science Department, University of Idaho*. Available at https://dl.acm.org/doi/10.5555/936065

[HX] Hwang J, Xie T, Hu V, Altunay M (2010) ACPT: A Tool for Modeling and Verifying Access Control Policies. *2010 IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2010)* (IEEE, Fairfax, VA), pp. 40-43. https://doi.org/10.1109/POLICY.2010.22

[IR7316] Hu VC, Ferraiolo DF, Kuhn DR (2006) Assessment of Access Control Systems. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 7316. https://doi.org/10.6028/NIST.IR.7316

[IR7874] Hu VC, Scarfone K (2012) Guidelines for Access Control System Evaluation Metrics (National Institute of Standards and Technology, Gaithersburg, MD), NIST Interagency or Internal Report (IR) 7874. https://doi.org/10.6028/NIST.IR.7874

[JO] Hu VC, Kuhn DR, Xie T, Hwand J (2011) Model Checking for Verification of Mandatory Access Control Models and Properties. *International Journal of Software Engineering and Knowledge Engineering* 21(1):103–127. https://doi.org/10.1142/S021819401100513X

[MO] Jayaraman K, Ganesh V, Tripunitara M, Rinard M, Chapin S (2011) Mohawk: Automatic Verification of Access-Control Policies. *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)* (ACM, Chicago, IL), pp. 163-174. https://dl.acm.org/doi/10.1145/2046707.2046727

[NU] Fondazione Bruno Kessler (2023) NuSMV: a new symbolic model checker. Available at https://nusmv.fbk.eu/

[NX] NetwokX developers (2024) NetworkX: Network Analysis in Python. Available at https://networkx.org/

[SF] NPTEL (2015) Safety properties described by automata. Available at https://www.youtube.com/watch?v=DzbqwlUw2-g

[SP] Spin developers (2024) Verifying Multi-threaded Software with Spin. Available at https://spinroot.com/spin/whatispin.html

[SP162] Hu VC, Ferraiolo DF, Kuhn DR, Schnitzer A, Sandlin K, Miller R, Scarfone KA (2014) Guide to Attribute Based Access Control (ABAC) Definition and Considerations. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-162, Includes updates as of August 02, 2019. https://doi.org/10.6028/NIST.SP.800-162

[SP192] Hu VC, Kuhn DR, Yaga D (2017) Verification and Test Methods for Access Control Policies/Models. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-192. https://doi.org/10.6028/NIST.SP.800-192

[WP] Wikipedia (2024) Dining philosophers problem. Available at https://en.wikipedia.org/wiki/Dining_philosophers_problem

[XA] OASIS Open (2023) XACML resources. Available at http://docs.oasis-open.org/xacml/

[YC] CTL* (2016) *NPTEL National Program on Technology Enhanced learning*. Available at https://www.youtube.com/watch?v=_2E5Q3CZ7g4&t=56s

[YC2] CTL (2016) *National Program on Technology Enhanced learning*. Available at https://www.youtube.com/watch?v=Blh060Hgbm8

[ZL] Potter B, Sinclair J, Till D (1996) An Introduction to Formal Specification and Z (Prentice Hall PTR, Upper Saddle River, NJ), 2nd Ed. Available at https://dl.acm.org/doi/10.5555/547639