International Journal of Information Security (2024) 23:3283–3310
https://doi.org/10.1007/s10207-024-00881-5

REGULAR CONTRIBUTION

The ACAC$_D$ model for mutable activity control and chain of dependencies in smart and connected systems

Tanjila Mawla[1] · Maanak Gupta[1] · Safwa Ameer[2] · Ravi Sandhu[2]

Published online: 20 July 2024

Abstract

With the integration of connected devices, artificial intelligence, and heterogeneous networks in IoT-driven cyber-physical systems, our society is evolving as a smart, automated, and connected community. In such dynamic and distributed environments, various operations are carried out considering different contextual factors to support the automation of connected devices and systems. These devices often perform long-lived operations or tasks (referred to as activities) to fulfill larger goals in the connected environment. These activities are usually mutable (change states) and interdependent. They can influence the execution of other activities in the ecosystem, requiring *active* and real-time monitoring of the entire connected environment. Traditional access control models are designed to take authorization decisions at the time of access request and do not fit well in dynamic and connected environments, which require continuous active checks on dependent and mutable activities. Recently, a vision for activity-centric access control (ACAC) was proposed to enable security modeling and enforcement from the perspective and abstraction of interdependent activities. The proposed ACAC incorporates four decision parameters: Authorizations (A), oBligations (B), Conditions (C), and activity Dependencies (D) for an *object agnostic* continuous access control in smart systems. In this paper, we take a step further towards maturing ACAC by focusing on the mutability of activities (the ability of changing states of activities), activity dependencies (D) and developing a family of formal mathematically grounded models, referred to as ACAC$_D$. We propose six practically suitable sub-models for ACAC$_D$ to support the state transition of a mutable activity incorporating the dependent activities' state-check and state-update procedures. These formal models consider the real-time mutability of activities as a critical factor in resolving *active* dependencies among various activities in the ecosystem. Activity dependencies can form a chain where it is possible to have dependencies of dependencies. In ACAC, we also consider the chain of dependencies while handling the mutability of an activity. We highlight the challenges (such as multiple dependency paths, race conditions, circular dependencies, and deadlocks) while dealing with a chain of dependencies, and provide solutions to resolve these challenges. We also present a proof of concept implementation of our proposed ACAC$_D$ models with performance analysis for a smart farming use case. This paper addresses the formal models' intended behavior while supporting activities' dependencies. Specifically, it focuses on developing and categorizing mathematically grounded activity dependencies into various ACAC sub-models without formal policy specification and analysis of theoretical complexities, which are intentionally kept out of the scope of this work.

Keywords Active access control · Activity control · Dependency · Mutability of activities · Smart and connected systems · Object agnostic · Chain of dependencies

✉ Tanjila Mawla
tmawla42@tntech.edu

Maanak Gupta
mgupta@tntech.edu

Safwa Ameer
safwa.ameer@gmail.com

Ravi Sandhu
ravi.sandhu@utsa.edu

[1] Department of Computer Science, Tennessee Tech University, Cookeville, TN, USA

[2] Institute for Cyber Security (ICS) and NSF C-SPECC Center, University of Texas at San Antonio, San Antonio, TX, USA

1 Introduction

Internet-of-Things (IoT) is a rapidly growing technology integrating billions of connected devices and artificial intelligence over heterogeneous networks, facilitating smart and collaborative ecosystems such as smart farming, smart manufacturing, smart cars, and e-health monitoring. In such dynamic and distributed environments, data-driven applications are widely used. Thousands of devices collect and utilize data from users, devices, and environments to support automation collaboratively. The ultimate goal of a futuristic community is to establish an autonomous smart ecosystem for human-driven domains where everything is connected, continuously communicating, sharing information, and triggering actions.

However, ensuring efficiency and accuracy for such systems while addressing growing security and privacy issues raises serious challenges in these smart communities' operational and administrative aspects. With increasing number of connected and interacting devices, the attack surface in such systems is continuously expanding. While cybersecurity is a top national priority and much progress has been made to ensure protection from cyber-attacks, IoT-driven smart systems security raises a host of new challenges. The convergence of the physical and cyber world introduces new automated attack dimensions which are hard to analyze, and engender substantial risk in maintaining the integrity of physical and cyber resources. Significant challenges to secure connected and IoT-driven systems include threat modeling, proposing mathematically grounded fundamental security approaches, continuous vulnerability assessment, and designing adaptable autonomous defense mechanisms to thwart rapidly evolving cyber-physical threats in this growing, connected, collaborative, and distributed ecosystem. These systems demand real-time **active** monitoring of operations and *activities* with the contextual information of multiple device states and environmental conditions for continuous authorization and system security. Access control solutions are extensively used to secure computer systems from unwanted and unauthorized access. Several traditional and extended access control solutions using discretionary, mandatory, role-based, or attribute-based approaches have been proposed to offer security needs for smart and connected systems [1–13]. However, traditional access control systems fall short in terms of dynamicity, scalability, mutability, and real-time monitoring needs of smart ecosystems. As we approach towards a fully automated, coordinated, data-driven, and highly connected future community supporting multi-domain/administered distributed collaborative devices, we need *active* access control models which can adapt to the dynamic context of the ecosystem, continuously monitor the changing access permissions and activities, and handle device failures while ensuring safety and security of the system.

In response, recently, Gupta and Sandhu [14] proposed a novel activity-centric access control (ACAC) paradigm supporting *activity* as the fundamental abstraction for the active run-time management of security in smart and collaborative systems. Intuitively an *activity* is a long-lived continuous event performed by a device in an automated system. Further, these activities change states as they progress and are also inter-dependent, i.e. an activity can control the execution of other activities in the ecosystem. In addition, these activities have chain of dependencies, meaning, an activity A is dependent on activity B, which in-turn is dependent on activity C, referred as *dependencies of dependencies*. Our previous work [15] proposed the integration of four decision parameters *Authorizations* (A), *oBligations* (B), *Conditions* (C) and *Dependencies* (D) in ACAC, as discussed in Sect. 2. Further, since smart systems have thousands of connected devices and frequent device failures, it is inefficient for a subject to decide (while making an access request) which particular device will perform the requested activity. In such cases, it is critical to shift to an *object-agnostic* model, where the system decides which object[1] is *best* to perform the activity, considering dependencies and other decision factors. This *object-agnostic* approach is very relevant in dynamic and scalable smart ecosystems where devices are randomly added or removed as the system scales. The goal is to approach security modeling and enforcement from the perspective (and abstraction) of activities and their dependencies in connected systems.

In this work, we propose a formal mathematically grounded family of ACAC models for activity dependencies (D), referred to as ACAC_D. We also show how these models can accommodate the chain of dependent activities providing solutions to some open problems. The main contributions of this paper are as follows.

- We motivate the need for *object-agnostic* access control which supports the mutability of dependent activities. We highlight the limitations of the existing access control models and distinguish ACAC in terms of dynamic activity dependencies, scalability, and activity mutability.
- We investigate the activity dependencies (D) component of the ACAC model. Toward this, we propose a family of six ACAC_D sub-models that cover pre-, post-, and ongoing dependencies.
- We provide formal definitions for ACAC_D sub-models and illustrate their intended behavior under different dependencies.

[1] Since, an activity is typically performed by an IoT device in smart ecosystems, we treat the terms object and device as equivalent in activity-centric access control.

- We investigate and analyze the chain of dependencies for a requested activity in different stages of its life cycle. We highlight the challenges of resolving the chain of dependencies and propose solutions.
- We demonstrate ACAC$_D$ sub-models with use case scenarios (including chain of dependencies) and present a proof of concept implementation to illustrate its application using commercially available technologies.

The rest of the paper is as follows. Section 2 motivates the need for activity-centric model, discusses the relevant background, and highlights the limitations of existing access control models. Section 3 presents our proposed family of ACAC$_D$ models with example use cases. Section 4 illustrates the challenges while resolving a chain of dependencies and shows how a combination of ACAC$_D$ sub-models are used to resolve a chain of dependencies. Section 5 provides a prototype implementation of ACAC$_D$ models and evaluates the performance with comprehensive smart farming use case. Section 6 discusses relevant literature on access control models and background. Section 7 concludes the paper.

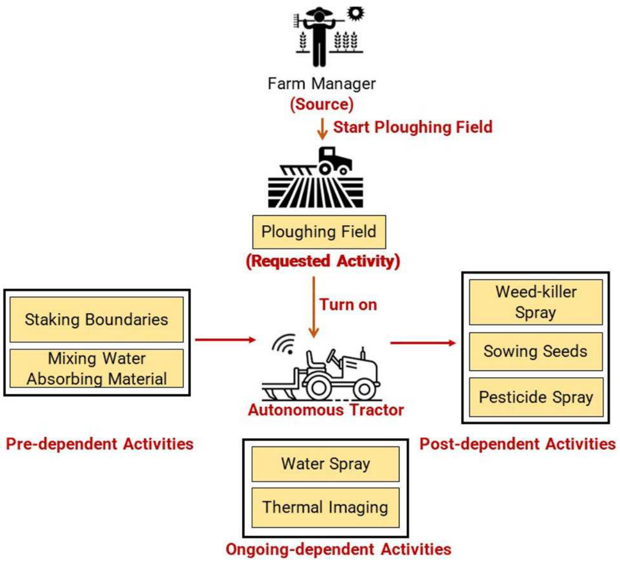


Fig. 1 The sets of pre-, ongoing and post-dependent activities with respect to a requested activity *ploughing field*. Each yellow box indicates an activity

2 Motivation for activity-centric "Active" access control

In smart and connected ecosystems, an activity is referred to as a long-lived continuous task that is performed by a device. At any given moment, thousands of activities and operations could be carried out depending on the workflow needs while considering related and different contextual factors. Activities in such systems are inter-dependent and can constrain the execution of each other. By an "Active" access control model for activity control, we refer to a security approach enforcing access control requirements where the system administrator or an automated system constantly monitors workflow needs, the state of the activity, and the decision (to initiate, continue, hold or revoke an activity) parameters. These decision parameters consist of authorizations, obligations, conditions, and dependencies on other activities. A user, device, or environmental event can request an activity based on the system workflow and efficiency needs. In general, the most suitable device can be assigned based on the decision parameters to satisfy the activity request.

In the example scenario shown in Fig. 1, an activity *ploughing field* is requested by a user *farm manager*. The system finds the most suitable device, which in our case is the *autonomous tractor*, to perform this requested activity. The corresponding operation, *turn-on* (calculated by the system based on the requested activity and selected device), is performed (if all decision parameters are satisfied) on behalf of the requesting source to initiate the activity *ploughing field*. However, whether the request is allowed or denied depends on the contextual information, including resolving the dependencies on various other activities in the system. As shown in the figure, there could be three sets of dependent activities; *pre-dependent* , *ongoing-dependent*, and *post-dependent*. Pre-dependent activities are checked before allowing the requested activity, ongoing-dependent activities are checked to ensure whether the execution of the requested activity can be continued or not (if dependencies are violated), and post-dependent activities are checked after the requested activity is revoked, on hold or finished. In this example, the requested activity *ploughing field* can be allowed only if the pre-dependent activities (*staking Boundaries, mixing water absorbing material*) are in their desired states. The continuity of the execution of the requested activity depends on the state of ongoing-dependent activities (*water spray* and *thermal imaging*). Finally, different post-dependent activities (*weed-killer spray, sowing seeds, pesticide spray*) are checked after the *ploughing field* activity is finished. The activities are mutable in nature, and can change their states (discussed in Sect. 3) to fulfill the dependency requirements. For example, an activity control policy can be that the *water spray* must be inactive while *ploughing field* is running. In such case, if *water spray* is running, it needs to transition to the finished or revoked state to ensure that it will be inactive immediately (if there is no post-dependent activity) to continue the activity *ploughing field*. In smart physical systems like smart farming that are either fully automated or semi-automated, the execution of dependent and composite activities necessitates minimal human intervention while maintaining the principle of least privilege. To maintain the system's performance, both safety and security are major concerns for smart and

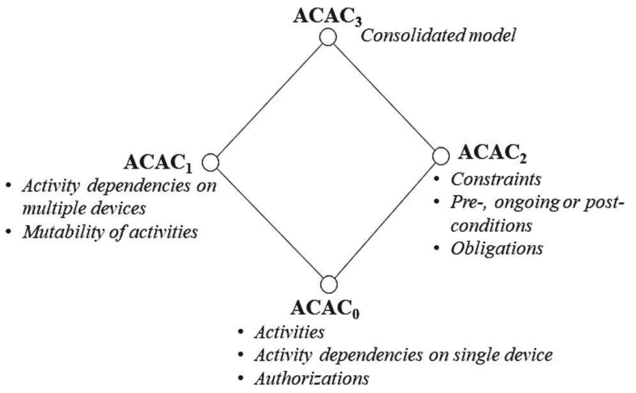

ACAC3
Consolidated model
ACAC1
• Activity dependencies on multiple devices
• Mutability of activities
ACAC2
• Constraints
• Pre-, ongoing or post-conditions
• Obligations
ACAC0
• Activities
• Activity dependencies on single device
• Authorizations

Fig. 2 A Framework for a Hierarchy of ACAC models [15]. $ACAC_0$, $ACAC_1$, and $ACAC_2$ models incrementally add features and create a final consolidated $ACAC_3$ model

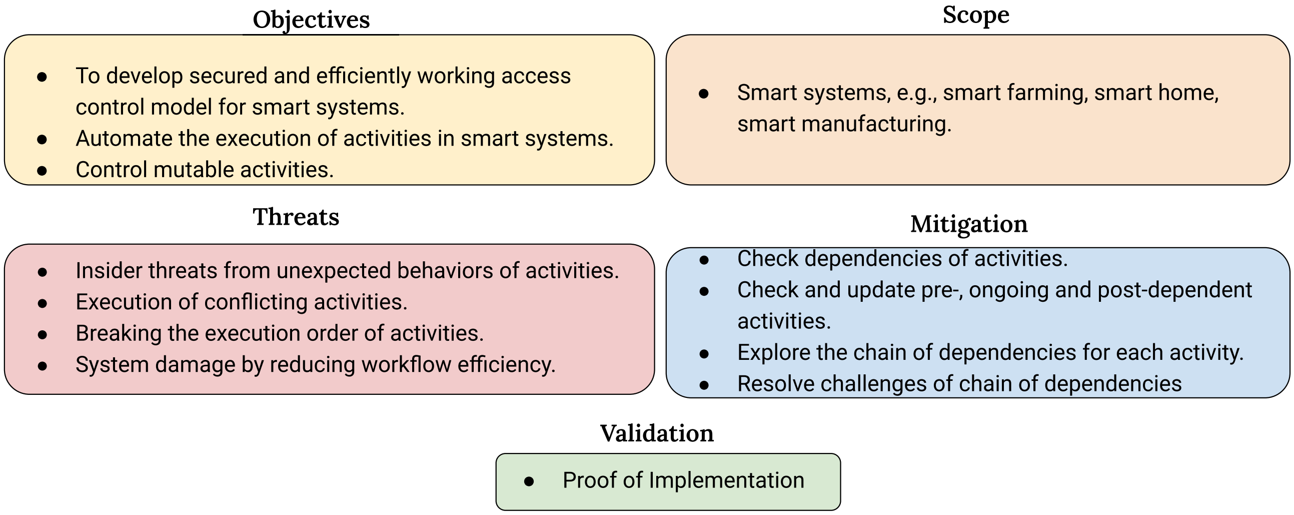
automated systems [16]. Several research works have been done where safety and security are analyzed to design intelligent and smart infrastructures such as smart grid [17], smart city [18], smart vehicles [19]. The current goal of building a smart community with technological advancement introduces various safety concerns that depend on what type of technologies are used and where it is being applied. While constructing an individual system activity irrespective of the other system activities, the goal of the system may have conflicts which in turn can create inherent loss or damage to the entire ecosystem. We are concerned about the safety of individual operation that is performed by IoT devices. Our goal is to build a safe system considering the probable interdependencies between the states of the different activities so that the system does not leave any gap between connected and dependent activities. For instance, protecting the system from breaking the order of activities, executing conflicting activities concurrently, and handling emergency activities hold major concerns from the safety aspect. In our proposed approach, the safe system we aim to build requires proper handling of activities throughout the life-cycle of an activity considering the environmental situation and relationships with other activities. Clearly, this approach requires continuous monitoring and real-time **active** dependency checks, making the ACAC novel and relevant for smart and collaborative ecosystems.

Recently, Mawla et al [15] proposed the components of the ACAC model and an incremental approach in a hierarchical framework to fully mature activity-centric access control. Instead of a monolithic model, different features are gradually added to a family of ACAC models, as illustrated in Fig. 2. The fundamental concept of activity and activity dependencies on a single device is captured in $ACAC_0$. In $ACAC_1$, activity dependencies on multiple devices and the mutability of activities are addressed. Note that, the activity dependencies on single or multiple devices are immaterial as ACAC is an *object-agnostic* model and considers security modeling at the *activity* abstraction. Therefore, both scenarios can be captured in $ACAC_1$ and the most suitable device is automatically decided by the system based on different factors. $ACAC_2$ adds static and dynamic constraints on activities, conditions (including system or environmental, e.g., weather, location), usage count, and obligations (required actions by the source). $ACAC_3$ is built on top of all ACAC models, which is the consolidated and detailed model to implement activity decision control in smart systems. Clearly, $ACAC_3$ will eventually cover the Authorizations (A), oBligations (B), Conditions (C) and Dependencies (D), as decision parameters, and can also be referred as $ACAC_{ABCD}$.

However, in this paper, we focus on the activity dependencies (D) component of ACAC. We develop formal mathematically grounded models for $ACAC_D$, which support the activity dependencies on multiple devices and the mutability of activities. We investigate the dependencies of dependencies to generate more fine-grained access control model. We also present a prototype implementation of our proposed family of $ACAC_D$ models and evaluate them using a comprehensive smart farming use case scenario with multiple activity requests and activity dependencies along with chain of dependencies.

2.1 Threat model

Figure 3 represents the threat model of our proposed $ACAC_D$ model. We follow the threat modeling steps proposed by OWASP [20]. This model is proposed based on activity dependencies in smart IoT-based systems where safety and security are the major concerns during the automation of different activities. Note that, the model acknowledges the presence of both immutable and mutable activities. Existing threats can exploit the vulnerabilities while the system wants to control the mutable activities according to the workflow preserving the safety of the system. In smart and connected systems, attacks can occur intentionally or accidentally by exploiting known and unknown vulnerabilities. Adversaries can be insiders or outsiders. Our primary emphasis is on insider threats that arise from unexpected behaviors, which can compromise system safety, violate workflows, and hinder efficiency. In complex systems with multiple devices performing various activities, a requester may not have knowledge of all the activities occurring. Consequently, simply checking authorization is insufficient for making activity decisions, as authorized users may still be restricted by activity dependencies. By considering these dependencies, we ensure the safety and security of the the system from conflicting activities, disruptions to the execution order, and violations of usage rules. Additionally, this approach enables the execution of emergency and high-priority activ-



Objectives

- To develop secured and efficiently working access control model for smart systems.
- Automate the execution of activities in smart systems.
- Control mutable activities.

Scope

- Smart systems, e.g., smart farming, smart home, smart manufacturing.

Threats

- Insider threats from unexpected behaviors of activities.
- Execution of conflicting activities.
- Breaking the execution order of activities.
- System damage by reducing workflow efficiency.

Mitigation

- Check dependencies of activities.
- Check and update pre-, ongoing and post-dependent activities.
- Explore the chain of dependencies for each activity.
- Resolve challenges of chain of dependencies

Validation

- Proof of Implementation

Fig. 3 Threat model for the proposed ACAC$_D$ model

ities. In our approach, denoted as ACAC$_D$, we assume that all sources are authorized for the requested activity, with a focus on verifying activity dependencies. We also take into account resolving the dependency chains to fulfill activity requests efficiently and in a secured way considering existing threats. However, we acknowledge the challenges involved in resolving these dependency chains and propose mitigation techniques. Our implementation serves as proof of the robustness of the ACAC$_D$ model.

2.2 Distinction from existing access control

In access control literature, different models (beyond classical DAC, MAC, and RBAC) have been proposed considering various decision parameters. Detailed in work by Mawla et al. [15], in this subsection, we review some of the closely related models with the ACAC model, Task-based Authorization Controls (TBAC) [21], Usage Control (UCON) [22], Activity-Centric Access Control for social computing (ACON) [23], Attribute-based Access Control (ABAC) [8, 24–26], and highlight key distinguishing features.

Table 1 summarizes the distinguishing features which are most relevant in terms of the notion of activity and activity-dependencies between ACAC and other models. The first column in the table contains the name of the models. The rest of the columns mention the key distinguishing features (we selected five, but could be more) among these models and if the models support these keys (Yes) or not (No). The key factors are *abstraction of activity*, *dynamic activity dependencies* (meaning activities are inter-dependent and dynamically calculated based on different factors), *object-agnostic* (refers that corresponding object for an activity will be decided by the system rather than by the requesting source at the time of request), *dependent activity mutability* (the property of changing dependent activity states), and *ongoing monitoring of the system context* (the system context information such as dependencies, usage, environmental conditions, etc., are continuously evaluated to support context-based access decisions).

Distinction from UCON: The proposed ACAC$_{ABCD}$ model is inspired by the UCON [22, 27] model. However, there are significant distinctions between UCON$_{ABC}$ and ACAC$_{ABCD}$ models. UCON supports attributes' mutability which is different from activity mutability supported by ACAC. UCON, primarily designed for digital rights management, does not have a notion of activity (which is a prolonged state of a device). In addition, UCON defines the object on which the operation is requested, which is different than ACAC$_{ABCD}$, which is an *object-agnostic* model. Further, the chain of dependencies supported in ACAC$_{ABCD}$ is not considered in UCON. The dependencies in ACAC can be on the same or different objects. Where the activity is actually executing or which source started the activity is irrelevant. The abstraction of activity in ACAC makes it easier to manage connected systems in terms of activities rather than objects and operations supported by UCON.

This comparison overview between ACAC and other related models strengthens the fact that how our proposed ACAC model distinctly supports 'active' decision control and enforcement considering dynamic situations and scalability in distributed IoT-based smart systems with thousands of connected devices performing multiple activities in a dynamic environment.

Table 1 Comparison of Features Proposed in ACAC Model

Access Control Models	Abstraction of activity	Dynamic activity dependencies	Object-agnostic	Dependent activity mutability	Ongoing monitoring of system context
TBAC	Yes	No	No	No	No
UCON	No	No	No	No	Yes
ACON	Yes	No	No	No	No
ABAC	No	No	No	No	No
ACAC	YES	YES	YES	YES	YES

3 Towards ACAC formal models

An **activity** is a prolonged event that is initiated by a source and occurs on an object for a certain period of time. The authors in [14, 15] motivated and proposed the activity-centric access control (ACAC) model components as shown in Fig. 4 and described as follows. A source (S) can be a device, sensor, user, or an event in the system that requests an activity. An activity (ACT) is a long continuous task occurring for a period of time. An object (O) is an entity that performs the activity, such as an IoT device. To start an activity, a source will perform an operation (OP) on the object. When a source requests to initiate an activity, the decision depends on four components: authorizations (A), obligations (B), conditions (C), and activity dependencies (D) in the system. Authorizations define the right of a source to initiate an operation on an object. Source and object attributes take part in the authorizations. Obligations are the required tasks that must be fulfilled by the same requesting source or a different source in the system. Conditions are system or environmental factors related to satisfying the requested activity. Dependency on activities reflects relationships between single or multiple device activities in a system. For example, in smart manufacturing, a robotic arm is requested to initiate *painting* a box. If the robotic arm is currently *washing* the product, it cannot be allowed immediately to paint the box. Here *painting* and *washing* are dependent activities. Our ultimate goal is to build an active security model for smart and collaborative systems utilizing all these components. However, with evolving different business needs and complexities, system designers and security administrators should be flexible in implementing some or all of these factors.

Accordingly, we define a family of four basic ACAC sub-models as ACAC$_{\text{A}}$, ACAC$_{\text{B}}$, ACAC$_{\text{C}}$, and ACAC$_{\text{D}}$ for the proposed consolidated ACAC model, referred as ACAC$_{\text{ABCD}}$. Each one of ACAC$_{\text{A}}$, ACAC$_{\text{B}}$, ACAC$_{\text{C}}$, and ACAC$_{\text{D}}$ is a family of models. ACAC$_{\text{A}}$ defines a family of models that define the authorization factor in a variety of ways to accommodate different application requirements. It considers the authorization factor only when deciding on an activity. ACAC$_{\text{B}}$ handles the obligations factor, ACAC$_{\text{C}}$ considers the impact of system and environmental conditions on an activity. ACAC$_{\text{D}}$ incorporates the dependencies between different activities in all stages of the life cycle of a requested activity by checking and updating the current states of the dependent activities.

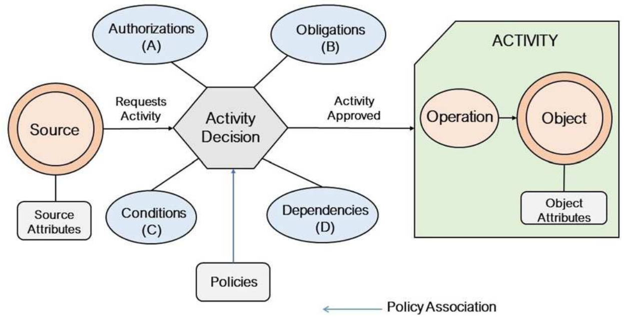


Fig. 4 **ACAC** Model **Components**: The source requests an activity. The activity decision components Authorizations (**A**), Obligations (**B**), Conditions (**C**) and Dependencies (**D**) on other activities are evaluated to allow or deny a requested activity. If allowed, the source performs an operation on the object to initiate the requested activity. Source and object attributes take part in the authorization. Policies are associated with the activity decision process [15]

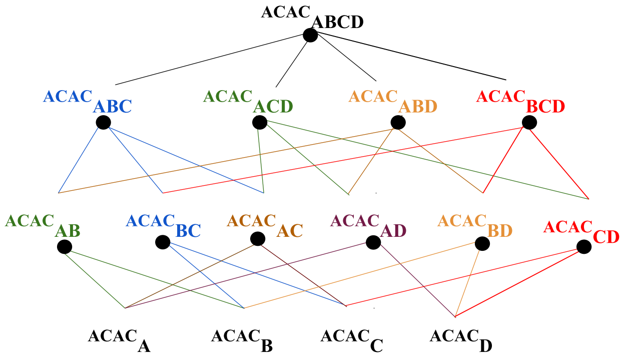


Fig. 5 The Combination of ACAC$_{\text{ABCD}}$ Core Models. The combination of the core models is created from the basic models (ACAC$_{\text{A}}$, ACAC$_{\text{B}}$, ACAC$_{\text{C}}$, and ACAC$_{\text{D}}$)

Our proposed ACAC$_{\text{ABCD}}$ model provides the **active** decision control by incorporating all of these decision factors [15]. Active decision control is defined as based on the real-time working environment considering authorizations, obligations, conditions, and dependencies on activities [15]. Considering the complexity, in Fig. 5, we show how the combination of ACAC$_{\text{ABCD}}$ core models are created from the basic models (ACAC$_{\text{A}}$, ACAC$_{\text{B}}$, ACAC$_{\text{C}}$, and ACAC$_{\text{D}}$). We put the basic models at the bottom level, which includes individual models for each decision component (A-B-C-D). At the next two levels, models are composed of two and three models, respectively, from the immediate lower levels. As shown in Fig. 5, ACAC$_{\text{ABCD}}$ is the final comprehensive model which combines the four sub-models. In order to consider the *active* security needs, in this paper, our focus is to develop formal sub-models for the dependency (D) factor considering the relationship of activities, referred to as

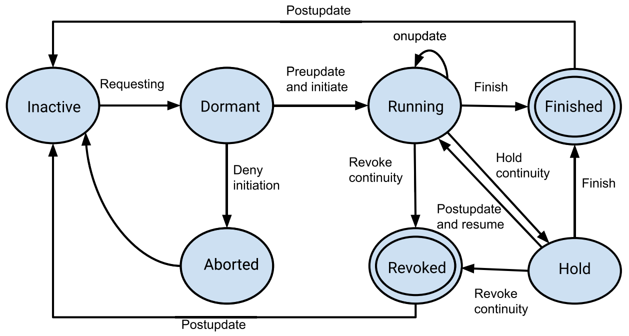


Fig. 6 State transition of an activity with required updates in the activity life-cycle. The blue shapes indicate activity states and the arrows pointing from one state to another indicate the activity transition from one state to another with necessary updates

ACAC$_D$ models. To our understanding and literature review, previous access control models have not considered these run-time dependencies as an active security factor, which is critical in smart connected and collaborative systems. The ACAC$_D$ is mapped to ACAC$_1$ model in the incrementally developed framework discussed by Mawla et al. [15]. In our future work, we will develop the holistic ACAC$_{ABCD}$ model considering the ACAC$_A$, ACAC$_B$, ACAC$_C$, and ACAC$_D$ basic models.

3.1 Mutability of activities

One of the ACAC model's unique characteristics is that the *activities* in the system are mutable. **Mutable** activities can update their states (as discussed by Mawla et al. [15]) as a consequence of the decision process of initiation, continuity, holding, completion, or revocation of an activity. In our models, **mutability** reflects the process of changing the state of mutable activities. In case of **immutable** activity, *no outside* factor can change the activity state, and activity will complete its task while transitioning within its pre-defined course of states. Figure 6 includes the states that an activity can have and shows the transitions between different states. An activity is in **inactive** state if it is not requested yet. When the activity is requested, the activity is in **dormant** state, and dependencies on other activities are assessed to see if the activity is allowed to be initiated. The dependent activities can be mutable and must change their states (if required) to allow the requested activity. In that case, the required pre-updates (updates before initiating an activity) on the dependent activities take place. Thus, the requested activity is invoked and goes to the **running** state. If the required pre-updates or any required condition cannot be fulfilled, the requested activity is denied and go to the **aborted** state. In the **running** state of activity, there can be required ongoing updates (updates during the execution of an activity to continue the execution) on the dependent activities. From the **running** state, an activity can be on **hold**, **finished**, or **revoked**. **Hold** state indicates a temporary suspension of the running activity due to any contextual conditions. Any required post update takes place after the activity goes to the **hold** state. From **hold** state the activity can resume and goes to the **running** state again. Otherwise, it can be **revoked** or **finished** based on the contextual conditions. The activity goes to a **revoked** state from the **running** state if the ongoing required updates (or ongoing conditions) are not fulfilled. **Finished** state indicates that the activity is completed and already served its purpose. Note that, from **finished** and **revoked** states, the requested activity goes back to the **inactive** state after the post-dependency check and update (if required). In Fig. 6, the names of the states are more intuitive which helps in a better understanding of an activity's life-cycle than shown in [15]. The transitions between activity states reflect the mutability of activities. It is a significant and distinguishable factor of ACAC compared with other access control models. In next subsection, we formally propose sub-models for ACAC$_D$ which considers the mutability of activities.

3.2 Chain of dependencies

A chain of dependencies refers to a series of dependencies where the dependency extends further down the line. In this paper, our goal is to inspect and analyze the dependent activities and control the mutability of these activities' states corresponding to a requested activity and its state transitions. In case the dependent activities, in-turn, have some dependencies, i.e. "*dependencies of dependencies*", we must ensure all the dependent activities are in their desired states before taking any decision on the requested activity. In such a scenario of a dependency chain, the system will wait to reach an independent activity (an activity that does not have any dependency) before any decision is made. We refer the requested activity as the "root" of the dependency chain and any dependent activity which depends on another activity for the state change is referred to as the "parent" of that dependent activity.

Figure 7 shows an example of a dependency chain corresponding to a requested activity "Sowing Seeds". This is requested by a Farm-manager and the system finds a "Seed-Drill" available to start "Sowing Seeds". Further, before allowing "Sowing Seeds" to start, we find two pre-dependent activities ("Water Pumping" and "Water Spraying") which are shown in the first level of dependency in the colored portion of the chain of dependent activities. The next level of dependent activities require to be in the desired states according to the current and desired states of the parent dependent activities. For instance, in the figure, "Nitrogen Spraying" is a dependent activity according to the current and desired state of "Water Spraying". In such scenarios with "dependencies of dependencies", we only can update the state of the parent

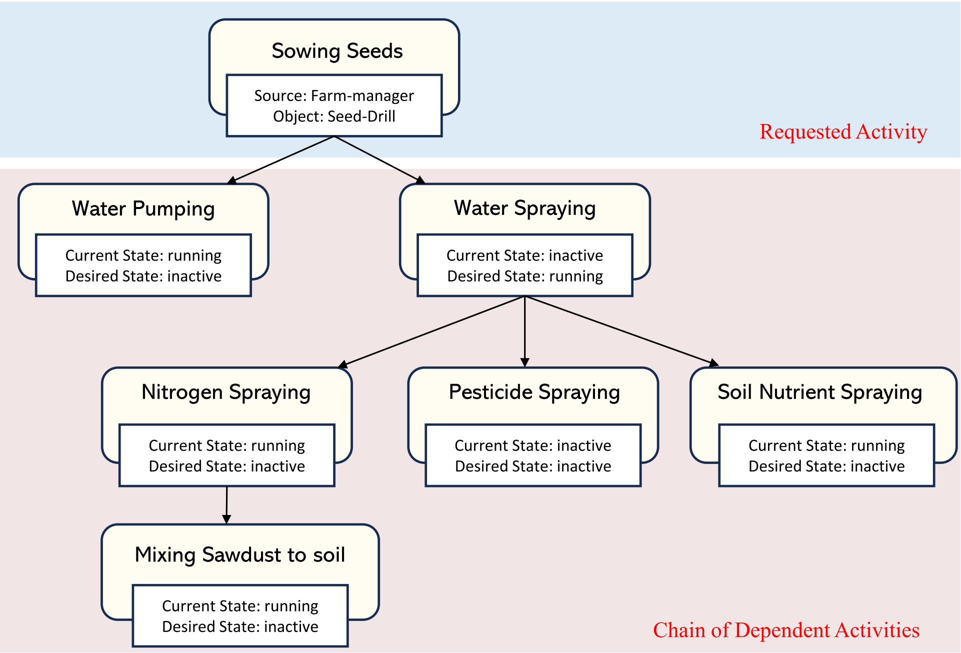


Fig. 7 Example of Chain of Dependencies. Yellow boxes represent activities. White box in "Requested Activity" includes the requesting source and a suitable object to perform activity *sowing seeds* while the white boxes in Chain of Dependent Activities include the current and desired states of each dependent activity. The arrows pointing from one activity to another indicate that the parent activity depends on the child activity to change its state from the current to the desired

Table 2 Family of ACAC$_{D}$ sub-models. The sub-models are created based on when the decision is made on the requested activity (indicated by preD and onD) and whether at each phase the model supports state-update of the dependent activities or not (indicated by 0, 1, 2, 3). 'Yes' indicates that the scenario is practical and 'No' indicates otherwise

	Immutable (0)	Pre-update (1)	Ongoing-update (2)	Post-update (3)
preD	Yes	Yes	No	Yes
onD	Yes	No	Yes	Yes

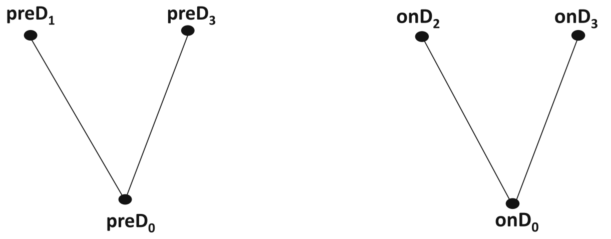
dependent activity when all dependent activities are in their desired states. This requires the system to find the chain of dependencies and update accordingly. In Section 4, we delve into the issue of the chain of dependencies. Throughout this section, we thoroughly examine the associated challenges and propose potential solutions to tackle this problem.

3.3 ACAC$_{D}$ formal models

Dependencies on activities (D) are created due to the relationships among activities. The activities can be on the same or different devices. As characterized by Gupta and Sandhu [14], related activities can be characterized as ordered, concurrent, temporary, precedent, dependent, conditional, and incompatible. In this paper, we are not trying to develop a policy language for ACAC$_{ABCD}$. Instead, we focus on formalizing the ACAC$_{D}$ models, which support the mutability of activities for active access control.

Table 2 shows the criteria for defining ACAC$_{D}$ sub-models. The models are classified based on two parameters: (a) When the dependencies on related activities are checked to take any decision on the requested activity. Decisions can be made *pre* i.e., before allowing the requested activity to start (referred to as preD) or *ongoing*, meaning while the requested activity is running (referred to as onD); (b) At which phase does the model support changing the states of dependent activities. The dependent activities can be either immutable or mutable, however, for immutable activities, the model cannot update the states and may result in activity request denial. We denote the case as '0' when the current and the desire states of the dependent activities are checked without supporting the updates on dependent activities. On the other hand, if the model supports changing the states of dependent activities, then state updates are possible before (pre), during (ongoing), or after (post) the requested activity is performed. These cases are denoted as '1', '2', and '3', respectively. In all cases, the dependent activities can be both immutable and mutable, however, updates on dependent activities can be possible in '1', '2', and '3' for mutable activities.

In Table 2, cases marked as 'Yes' indicate the more practical scenarios considering when a decision is made, and when dependent activities change state. Cases marked by 'No' indicate that such scenarios are not practically useful. If the decision is taken before allowing the requested activity, updates on the dependent activities can occur before (pre) and after (post) the requested activity is performed. Without ongoing-decision, there is no need to have ongoing-update as a part of mutability, and is thus marked as 'No'. For example, dependent activity B must be started before allowing requested activity A to start, and B should be revoked after A is finished. This case can be handled using pre and post



(a) Sub-models for **preD** (b) Sub-models for **onD**

Fig. 8 Categorization of ACAC$_D$ sub-models

update of B as a consequence of the initiation of A, and does not require ongoing-updates on B. However, if the decision is taken while the requested activity is ongoing, updates on the dependent activities can occur during (ongoing) and after (post) the requested activity is performed. In case of an ongoing-decision, the activity is already initiated. Thus, onD does not consider the pre-updates on the dependent activities and is marked as 'No' for pre-update (1) of the onD case. The six 'Yes's in Table 2 define six basic ACAC$_D$ sub-models, which will be formalized in the following sections.

Different sub-model combinations of ACAC$_D$ will be required for different type (pre, ongoing or post) of updates to solve all the recursive dependencies, as each sub-model defines a specific type of update. Further, the dependent activities can be on the same or different device on which the activity is requested. Moreover, the dependent activity may be initiated by different objects in the system. In our model, the system chooses the object which can fulfil the activity, as will be discussed in the following sections.

In Fig. 8, we show how the family of ACAC$_D$ model is categorized into different sub-models. The '0' cases for both preD and onD models only support checking the current and desired states of the activities, without any state updates. The '1', '2', and '3' cases supporting mutability add update procedures for the dependent mutable activities, and thus, inherit the basic components from the corresponding '0' cases. It should be noted that if the dependent activity is immutable, no state updates are allowed, and will result in activity request denial if the current and the desired states do not match. We formally discuss the components for each sub-model in the following subsection.

In real-world use-cases, the activity-centric approach may need a **combination of two or more** ACAC$_D$ **sub-models** checking pre-, ongoing, and post-dependencies. However, for clarity, we will formalize the behavior of the sub-models individually, and in our prototype implementation in Sect. 5, we experiment with a more holistic multi-model comprehensive use case scenario.

Table 3 elaborates the basic sets and functions we use in the formal definitions (1-6) and Algorithms (1,2,3). S, O, and OP are the finite sets of sources, objects, and operations in the system [in Fig. 4, source is shown in a circle in the left part, and operation and object are shown respectively in elliptical and circle shape in the green part]. ACT is a finite set of activities that can be performed in the system. ACT_R, ACT_D, ACT_{DoD} are the finite sets of requested activities, dependent activities, and dependent of dependent activities respectively which are equivalent to the set of activities, ACT, formally we can say $ACT_R = ACT_D = ACT_{DoD} = ACT$. ST is the finite set of the activity states which is defined in the system as $\{inactive, dormant, aborted, running, hold, revoked, finished\}$. ST_{CR} and ST_{DR} are the finite set of current and desired states which are equivalent to the set, ST, formally we can say $ST_{CR} = ST_{DR} = ST$. The function $getObject$ maps a requested activity to the most suitable object to perform the activity in the system. This function can be called using a requested activity $act \in ACT$ and provides the most suitable object $o \in O$. $getOperation$ function determines the corresponding operation to start the requested activity on the chosen object by the system. $getDA$ function maps a requested activity and its corresponding object to a set of dependent activities (ACT_D). The dependent activities for a particular requested activity can vary depending on the corresponding object. *getCurrentSt* function maps an activity to a current state. *assignedDesiredSt* function maps a dependent activity to an empty set or a desired state. This function is used to store a currently assigned desired state for a dependent activity. *getBinarySemaphoreValue* function is used to provide the currently assigned value (0 or 1) for a dependent activity meaning that this activity is locked (cannot change the state) or unlocked by another activity. *hasConflictingDesiredSt* function maps a dependent activity to TRUE or FALSE meaning whether that dependent activity has conflicting (multiple) desired states or not. $getDoDA$ function takes the input of a dependent activity, the current and desired state of this activity, and provides a set of activities which we call dependent of dependent activities. We refer 'DoD' subscript to "dependent of dependent". To get the desired state of a dependent of dependent activity, we use the function $getDesiredDoDASt$ which maps a dependent activity, its current and desired state and a dependent of this dependent activity to a desired state. Apart from the basic sets and function in Table 3, we use two more functions from the algorithms (elaborated in Sect. 4) in the model definitions. One is RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION(da, $da\_current\_st$, $da\_desired\_st$), which is a function in Algorithm 1 that recursively checks if an activity, da has dependencies to transition from $da\_current\_st$ to $da\_desired\_st$ and for each activity, it detects whether the activity has conflicting desired states (multiple desired states) or not and stores the information. Another function is RECURSIVE-UPDATE(da, $da\_current\_st$, $da\_desired\_st$) function from Algorithm 2 which recursively handles the

Table 3 Introduction to basic sets and functions used in the definitions and algorithm

Basic Sets	**Description**
S, O, OP, ACT	Finite sets of sources, objects, operations, and activities in the system, respectively.
ACT_R, ACT_D	Sets of requested and dependent activities such that $ACT_R = ACT_D = ACT$.
ACT_{DoD}	Set of dependent of dependent activities, where $ACT_{DoD} = ACT$.
ST	Finite set of states of activities, where $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$.
ST_{CR}, ST_{DR}	Finite sets of current and desired states of an activity, where $ST_{CR} = ST_{DR} = ST$.
Common Functions for Definitions	**Description**
$getObject : ACT_R \longrightarrow O$	Mapping requested activity to an object.
$getOperation : ACT_R \times O \longrightarrow OP$	Mapping a requested activity and an appropriate object to an operation to execute the activity.
$getDA : ACT_R \times O \longrightarrow 2^{ACT_D}$	Mapping a requested activity and an object to a set of dependent activities.
Common Functions for Definitions and Algorithms	**Description**
$getCurrentSt : ACT \longrightarrow ST_{CR}$	Mapping an activity to its current state.
Algorithm Functions	**Description**
$assignedDesiredSt : ACT_D \longrightarrow \{\emptyset, ST_{DR}\}$	Mapping a dependent activity to a currently assigned desired state or an empty set (meaning there is no desired state assigned yet).
$getBinarySemaphoreValue : ACT_D \longrightarrow \{0, 1\}$	Mapping from ACT_D to $\{0, 1\}$, 0 and 1 respectively indicate that the input dependent activity is currently locked and unlocked.
$hasConflictingDesiredSt : ACT_D \longrightarrow \{TRUE, FALSE\}$	$TRUE$ indicates the input dependent activity has conflicting (multiple) desired states and $FALSE$ indicates it has no conflicting desired state.
$getDoDA : ACT_D \times ST_{CR} \times ST_{DR} \longrightarrow 2^{ACT_{DoD}}$	Mapping a dependent activity, the current state of the dependent activity and a desired state of the dependent activity to a set of dependent of dependent activities.
$getDesiredDoDASt : ACT_D \times ST_{CR} \times ST_{DR} \times ACT_{DoD} \longrightarrow ST_{DR}$	Mapping a dependent activity, the current state of dependent activity, desired state of dependent activity, and a dependent of dependent activity to a desired state.

state check and update process for all dependencies (including chain of dependencies) of a dependent activity, da.

3.3.1 ACAC$_{\text{preD}}$ - pre-dependency models

ACAC$_{\text{preD}}$ models utilize the dependencies related to the decision process before the initiation of the requested activity. ACAC$_{\text{preD}}$ has three sub-models (stated in Fig. 8a ACAC$_{\text{preD}_0}$ model checks the pre-dependencies that are required to allow the requested activity. ACAC$_{\text{preD}_0}$ model does not support mutability (i.e. cannot update dependent activity states). ACAC$_{\text{preD}_1}$ model allows pre-updates on the dependent activities that require to be in specific states to allow the requested activity. ACAC$_{\text{preD}}$ does not have ongoing-update model since ongoing-update without ongoing-decision does not need to be considered as a part of mutability. Post-updates on dependent activities as a consequence of the pre-decision process are handled in ACAC$_{\text{preD}_3}$ model. The following three definitions formalize ACAC$_{\text{preD}}$ models. We elaborate the basic sets and functions in Table 3 and use the necessary sets and functions in these definitions from the table.

Definition 1. ACAC$_{\text{preD}_0}$: **Pre-dependency checking model for pre-dependent activities.** ACAC$_{\text{preD}_0}$ model checks the current and desired states of the pre-dependent activities before allowing a requested activity. This model does not have any update procedure for state change and cannot support mutability of dependent activities. ACAC$_{\text{preD}_0}$ consists of the following components (shown in Fig. 4), and explained later:

– S, O, OP, ACT, ACT_R, ACT_D, ST, ST_{CR}, ST_{DR} are finite sets of sources, objects, operations, activities, requested activities, dependent activities, activities' states, current states and desired states respectively [elaborated in Table 3]. A source $s \in S$ requests to perform an activity $act \in ACT$, defined as $request(s, act)$. To satisfy this activity request (formally stated as, $request(s, act) = True$), the system will first specify an appropriate object $o \in O$, and perform an operation $op \in OP$ (Note that, whether source s is allowed to perform an operation op on an object o is determined by the authorization model $\mathrm{ACAC_A}$). Then, the system will check activity dependencies based on the corresponding to the requested activity and the object, using the $getDA$ function.

- $getDesiredPreDASt$: $ACT_R \times ACT_D \longrightarrow ST_{DR}$ ▷ ***[mapping a requested activity, and a dependent activity to a desired state.]***
- $preD(act\text{: } ACT,\ o\text{: } O) \longrightarrow \{True,\ False\}$, defined as $\bigwedge_{(da \in getDA(act,o))} getCurrentSt(da) = getDesiredPreDASt(act, da)$
- $allowed(s\text{:}S,\ o\text{: } O,\ op\text{: } OP,\ act\text{: } ACT) \Rightarrow preD(act, o)$

$\mathrm{ACAC_{preD_0}}$ model consists of sources (S), objects (O), operations (OP), activities (ACT), requested activities (ACT_R), dependent activities (ACT_D), finite set of activities' states (ST), activities' current states (ST_{CR}) and activities' desired states (ST_{DR}). The function $getObject$ maps a requested activity to the most suitable object $o \in O$ to perform the activity in the system. $getOperation$ determines the corresponding operation to start an activity on the chosen object, o. More than one combination of activity and object can be mapped to an operation. The function $getDA$ computes the set of dependent activities, decided based on the activity $act \in ACT$ and the corresponding object $o \in O$. Note that the dependencies are dynamic, and can change based on conditions (C) and contextual factors. This is a many-to-one mapping function where each combination of activity and object can be mapped to a set of activities. The function $getCurrentSt$ is used to get the current state of an activity and $getDesiredPreDASt$ is used to determine the desired states of pre-dependent activities (activities that need to be checked before starting activity act). $getCurrentSt$ and $getDesiredPreDASt$ are many-to one mapping functions.

$preD$ is a functional predicate that takes the requested activity and the corresponding object (since dependencies can change based on which object is performing the activity) as inputs, and return $True$ or $False$ by comparing the current and desired states of all pre-dependent activities. $True$ indicates that all dependent activities' current states are in the desired states. $False$ indicates that at least one dependent activity is not in the desired state to allow the requested activity to be initiated. To allow the request, formally stated as $request(s, act) = True$, the $allowed(s, o, op, act)$ function (which decides s can perform operation op to start the activity act on the object o) should evaluate to $True$. The $allowed$ function returns $True$ if $preD$ evaluates to $True$. Note that, we use the $implies$ ($\Longrightarrow$) connective where the right hand side of the connective is necessary but not sufficient since authorization (A), oBligations (B) and conditions (C) also be checked for the left hand side to be $True$. There is no update procedure in this model.

Example 1. In smart manufacturing, a $robot$ is trying to make a $forceGeneration$ activity request, stated as $request(robot, forceGeneration)$.

- $S = \{robot\}$
- $O = \{motor\}$
- $OP = \{turnOn, turnOff\}$
- $ACT = \{forceGeneration, vibrationMonitoring\}$
- $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$
- $getObject(forceGeneration) = motor$
- $getOperation(forceGeneration, motor) = turnOn$
- $getDA(forceGeneration, motor) = \{vibrationMonitoring\}$
- $getCurrentSt(vibrationMonitoring) = running$
- $getDesiredPreDASt(forceGeneration,\ vibrationMonitoring) = running$
- $preD(forceGeneration, motor) = True$
- $allowed(robot,\ motor,\ turnOn,\ forceGeneration) \Rightarrow preD(forceGeneration, motor)$

In this example, to satisfy the request made by the source $robot$, we get the corresponding object $motor$ and operation $turnOn$ for the requested activity. The set of dependent activities for $forceGeneration$ consists of $vibrationMonitoring$. The desired state of $vibrationMonitoring$ is $running$. In this instance, the current state is same as the desired state for the only dependent activity. Thus, $preD(forceGeneration, motor)$ is $True$ as the necessary condition (comparing the current and desired states of the dependent activity) in $preD(forceGeneration, motor)$ is fulfilled. The $allowed$ function also returns $True$ which decides that source $robot$ is allowed to perform the operation, $turnOn$ on the object $motor$ to initiate the requested activity, $forceGeneration$.

Definition 2. $\mathrm{ACAC_{preD_1}}$: **Pre-update model for pre-dependent activities.** $\mathrm{ACAC_{preD_1}}$ model adds state update procedure for the pre-dependent activities (dependent activities that are required to be in desired state before initiation of the requested activity). These pre-dependent activities may, in-turn, be dependent on other activities. For example, starting the requested activity A depends on starting the dependent activity B. Activity B can't start until activity C has already started. In such situations, we have to update the states of the pre-dependent activities in a recursive way, where we explore the "*dependencies of dependencies*" until we find a dependent activity that does not have any dependent activity before changing its state or all dependent activities need to be already in their desired states. Algorithm 1 includes a function named RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION where a dependent activity, the current and desired state of that activity are passed as parameters. We check if this dependent activity has any conflicting (multiple) desired states or not and store this information. Note that, this function is recursive and we recursively detect the conflicting desired states for all "dependencies of dependencies" along with the dependent activity (explained in Sect. 4). In Algorithm 2 in Sect. 4, we have a function named RECURSIVE-UPDATE. In this function, we pass the parameters for a dependent activity, its current state and a desired state of this dependent activity. This function returns the desired state after checking and updating (if necessary) all the "dependencies of dependencies". We explain Algorithm 2 in Sect. 4 describing the way it works with the recursive update procedure of "chain of dependencies". Conceptually, $\text{ACAC}_{\text{preD}_1}$ model is an extension to $\text{ACAC}_{\text{preD}_0}$ as it adds the pre-update procedure when *allowed* function returns False. Thus, to satisfy the activity request $request(s : S, act : ACT) = True$, $\text{ACAC}_{\text{preD}_1}$ model allows updating the states of the pre-dependent activities using the following *preUpdate*(act) function defined as.

- *preUpdate*(act, o): ▷ ***[Function Definition]***

  $(\forall da \in getDA(act, o))$.
  [RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION$(da, getCurrentSt(da), getDesiredPreDASt(act, da))$

  $getCurrentSt(da) \neq getDesiredPreDASt(act, da) \Rightarrow getCurrentSt(da) =$ RECURSIVE-UPDATE$(da, getCurrentSt(da), getDesiredPreDASt(act, da))$]
- *preUpdate*$(act, o) \Rightarrow allowed(s, o, op, act) == False$ ▷ ***[Function Call]***

$\text{ACAC}_{\text{preD}_1}$ model introduces the *preUpdate* function to update the states of the pre-dependent activities that are required to be in specific states for the initiation of the requested activity *act* on the object *o*. In this function, we iterate a loop for all the dependent activities where the current state of each dependent activity is updated to the desired state if it is not in the desired state at the time of the request. Before updating the current state of each dependent activity, we check whether the dependent activity (including its dependencies) in the loop has conflicting desired states or not utilizing the function, RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION in Algorithm 1. After that, we call the RECURSIVE-UPDATE function in Algorithm 2 by the dependent activity, its current state, and the desired state and resolve the state-updates for "chain of dependencies" where it is required. This function returns the desired state and we update the current state to the desired state. *preUpdate* function is called when the *allowed* function returns $False$ as the current states of all the dependent activities are not in their desired states. For simplicity, issues like who will update the state of the activity and underlying technical implementation of the update procedure is left unspecified in this paper.

Example 2. In smart home, the $houseOwner$ is trying to make the request for the activity, $playingNews$. The request is stated as $request(houseOwner, playingNews)$.

- $S = \{houseOwner\}$
- $O = \{TV, googleHome\}$
- $OP = \{turnOn, turnOff\}$
- $ACT = \{playingSong, playingNews\}$
- $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$
- $getObject(playingNews) = TV$
- $getOperation(playingNews, TV) = turnOn$
- $getDA(playingNews, TV) = \{playingSong\}$
- $getCurrentSt(playingSong) = running$
- $getDesiredPreDASt(playingNews, playingSong) = inactive$
- $preD(playingNews, TV) =$ False
- $allowed(houseOwner, TV, turnOn, playingNews) \Rightarrow preD(playingNews, TV)$
- *preUpdate*$(playingNews) \Rightarrow preD(playingNews, TV) == False$

In Example 2, to satisfy $request(houseOwner, playingNews)$, we get the corresponding object TV and the operation $turnOn$. The set of dependent activities (provided by $getDA(playingNews, TV)$) for $playingNews$ consists of $playingSong$. In this instance, the current state of $playingSong$ is *running*, which is not the same as the desired state *inactive*. Thus, $preD$ is false, and so is the *allowed* function. Therefore, the model updates the current state of $playingSong$ to *inactive* using the *preUpdate*$(playingNews)$ function. Once updated, the req uest $request(houseOwner, playingNews)$ is allowed.

Definition 3. $\text{ACAC}_{\text{preD}_3}$: **Post-update model for dependent activities with pre-check.** $\text{ACAC}_{\text{preD}_3}$ model adds the post-update procedure which updates the states of the dependent activities after the requested activity is finished, revoked or on hold. Updating the states of these dependent activities accumulate the consequence of the requested activity. In pre-check, we check the pre-dependent activities that need to change their states after the completion or revocation of the requested activity. For example, a dependent activity B have already started to help executing the requested activity A. After A is finished, activity B is no longer needed. Thus, we make sure there are no unnecessary activities going on after the purpose is completed. In such cases, combination of pre-update and post-update models is more appropriate. However, we consider post-update as a separate procedure. Conceptually, ACAC$_{\mathrm{preD}_3}$ model is an extension to ACAC$_{\mathrm{preD}_0}$ which adds the post-update procedure.

- $getDesiredPostDASt$: $ACT_R \times ACT_D \longrightarrow ST_{DR}$ ▷ ***[mapping a requested activity which has either been on 'hold', 'finished' or 'revoked', and a post-dependent activity to a desired state]***

- $postD(act\colon ACT,\ o\colon O) \longrightarrow \{True,\ False\}$, defined as $\bigwedge_{(da \in getDA(act,o))} getCurrentSt(da) = getDesiredPostDASt(act, da)$

- *postUpdate*(act, o): ▷ ***[Function Definition]***
  $(\forall da \in getDA(act, o))$.
  [RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION(da,$getCurrentSt(da)$, $getDesiredPostDASt(act, da)$)

  $getCurrentSt(da) \neq getDesiredPostDASt(act, da) \Rightarrow getCurrentSt(da)$ = RECURSIVE-UPDATE(da, $getCurrentSt(da)$, $getDesiredPostDASt(act, da)$)
  ]
- *postUpdate*$(act, o) \Rightarrow postD(act, o)$ == False ▷ ***[Function call]***

ACAC$_{\mathrm{preD}_3}$ model includes the *postUpdate* function to update the states of the dependent activities after the requested activity act is performed. The $getDesiredPostDASt$ is a many-to-one function to get the desired states of the post-dependent activities. It maps the requested activity and a dependent activity to a desired state. Then the $postD$ function is evaluated checking the current and desired states of the post-dependent activities. In *postUpdate* function, conflicting desired states are checked for all the post-dependent activities calling the RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION function from Algorithm 1 followed by updating their current states to their corresponding desired states utilizing the RECURSIVE-UPDATE function from Algorithm 2. This *postUpdate* function is called when $postD$ returns $False$ (which means that the current states of all dependent activities are not in their desired states).

Example 3. In smart industry, a $productionWorker$ is requesting $hydrotreating$ activity, formally stated as $request(productionWorker, hydrotreating)$.

- $S = \{productionWorker\}$
- $O = \{tankPump, hydrotreater\}$
- $OP = \{turnOn\}$
- $ACT = \{oilPumping, hydrotreating\}$
- $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$
- $getOperation(hydrotreating,\ hydrotreater) = turnOn$
- $getDA(hydrotreating,\ hydrotreater) = \{oilPumping\}$
- $getCurrentSt(oilPumping) = inactive$
- $getDesiredPostDASt(hydrotreating, oilPumping) = running$
- $postD(hydrotreating, hydrotreater)$ = False
- *postUpdate*$(hydrotreating) \Rightarrow postD(hydrotreating, hydrotreater) == False$

In Example 3, the requested activity is $hydrotreating$. This request was allowed and has just finished. Now, we need to update the post-dependent activities of $hydrotreating$. We get the set of dependent activities for $hydrotreating$ (using $getDA(hydrotreating, hydrotreater)$ function) which consists of one activity, $oilPumping$ (assuming $oilPumping$ already served its purpose of activating $hydrotreating$). The current and desired states of $oilPumping$ are not same in this instance. Thus, the $postD$ function returns $False$. We call *postUpdate*$(hydrotreating)$ function where the current state of $oilPumping$ is updated to the desired state.

3.3.2 ACAC$_{\mathrm{onD}}$ - ongoing-dependency models

ACAC$_{\mathrm{onD}}$ models consider the dependencies on activities while the requested activity is ongoing. The ongoing decisions can be *continue*, *hold* or, *revoke* the requested activity, and can impact dependent activities. Execution of the requested activity can be *continued* if the ongoing dependent activities are in the desired states. If the dependent activities are mutable, their current states can be updated for the continuity of the requested activity. Otherwise, the execution of the requested activity will be *revoked*. Besides that, holding the requested activity can accumulate any emergence or contextual situations. ACAC$_{\mathrm{onD}}$ has three sub-models (stated in Fig. 8b) based on if states of dependent activities can be updated and which phase the updates can occur as shown in Table 2. ACAC$_{\mathrm{onD}_0}$ model checks the current and desired states of the ongoing dependent activities. ACAC$_{\mathrm{onD}_0}$ model does not support mutability. ACAC$_{\mathrm{onD}_2}$ allows updates on the states of the ongoing dependent activities as a consequence of the ongoing-decisions. ACAC$_{\mathrm{onD}_3}$ model checks and updates the post-dependent activity states that are related to the ongoing activity and decisions. ACAC$_{\mathrm{onD}}$ does not have the ACAC$_{\mathrm{onD_1}}$ model since the requested activity is already allowed and there is no reason to consider the pre-updates after allowing the activity. Since the ongoing dependent activities are checked during the execution of the requested activity, how frequently the dependencies are checked is unspecified, and left for the implementation details.

Definition 4. ACAC$_{\mathrm{onD_0}}$: **Ongoing-dependency checking model for ongoing dependent activities**
ACAC$_{\mathrm{onD_0}}$ model checks the dependencies on activities while the requested activity is running to decide *continuity* or *revocation* of the ongoing activity. There is no update procedure in this model. We need this model only to check if all the ongoing dependent activities are in their desired states or not. The model consists of the following components:

A source $s \in S$ requests to perform an activity $act \in ACT$, defined as $request(s, act)$. Since, ACAC$_{\mathrm{onD_0}}$ model checks the ongoing dependencies on activities, the requested activity is assumed to be initially allowed.

- $allowed(s{:}\ S, o{:}\ O, op{:}\ OP, act{:}\ ACT) \Rightarrow True$
- $getDesiredOnDASt{:}\ ACT_R \times ACT_D \longrightarrow ST_{DR}$ ▷ ***[mapping a requested 'running' activity, and an ongoing-dependent activity to a desired state.]***
- $onD(act{:}\ ACT, o{:}\ O) \longrightarrow \{True, False\}$, defined as $\bigwedge_{(da \in getDA(act,o))} getCurrentSt(da) = getDesiredOnDASt(act, da)$
- $stopped(act{:}\ ACT, o{:}\ O) \Rightarrow onD(act, o) == False$ ▷ ***[Function call]***

ACAC$_{\mathrm{onD_0}}$ model consists of sources (S), objects (O), operations (OP), activities (ACT), requested activities (ACT_R), dependent activities (ACT_D), finite set of activities' states (ST), activities' current states (ST_{CR}) and activities' desired states (ST_{DR}) [explained in Table 3]. $getObject$ function provides the corresponding object the activity is running on. $getOperation$ function provides the operation *op* that is performed on object *o* to initiate the requested activity, *act*. The *allowed* function is $True$ since the requested activity is already assumed to be running currently, and the check is only made for ongoing decision.

$getDA$ function computes the set of dependent activities for the ongoing activity, $act \in ACT$. $getDesiredOnDASt$ is used to get the desired states of the ongoing-dependent activities. This function maps the requested 'running' activity and a dependent activity to a desired state. onD is a functional predicate which takes input of the requested activity and corresponding object (since dependencies can change based on the object which is performing the activity), and compares the current and desired states of all ongoing-dependent activities (and returns $True$ or $False$) to make a decision. Ongoing dependencies are checked throughout the execution of the activity act using the onD function. If onD returns $False$, the activity will be revoked which is handled using the $stopped$ function. We do not have any update procedure in this model.

Example 4. In smart farming, activity $cooling$ is requested by the $farmManager$ (formally stated as $request(farmManager, cooling)$) and is assumed to be allowed. In the ongoing check, our model ensures the corresponding dependencies are fulfilled.

- $S = \{farmManager\}$
- $O = \{cooler, aerialDrone\}$
- $OP = \{turnOff, turnOn\}$
- $ACT = \{thermalImaging, cooling\}$
- $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$
- $getObject(cooling) = cooler$
- $getOperation(cooling, cooler) = turnOn$
- $getDA(cooling, cooler) = \{thermalImaging\}$
- $getCurrentSt(thermalImaging) = inactive$
- $getDesiredOnDASt(cooling, \quad thermalImaging) = running$
- $onD(cooling, cooler) = False$
- $stopped(cooling, cooler)$

In this example, $thermalImaging$ is an immutable and ongoing-dependent activity for $cooling$ to obtain the current temperature and relevant status of the environment. The desired state of $thermalImaging$ is $running$ to continue $cooling$. As the current state of $thermalImaging$ is $inactive$ (and cannot be changed) which is different from the desired state, $cooling$ will be revoked.

Definition 5. ACAC$_{\mathrm{onD_2}}$: **Ongoing-update model for ongoing dependent activities**
ACAC$_{\mathrm{onD_2}}$ model adds the update procedure to change the states (if not in desired state) of the ongoing dependent activities of a requested activity. The updates are required to allow the requested activity to *continue*. For example, A is the requested activity which is executing and B is the dependent activity that should be $running$ to continue activity A. In this model, we can update the state of activity B from $inactive$ to $running$ to allow the activity A to *continue*. ACAC$_{\mathrm{onD_2}}$ model includes a function *onUpdate* for such ongoing updates. This model is an extension to ACAC$_{\mathrm{onD_0}}$ adding the ongoing update procedure.

- *onUpdate*(act, o): ▷ ***[Function Definition]***
  $(\forall da \in getDA(act, o))$.
  [RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION$(da, \quad getCurrentSt(da),$

$getDesiredOnDASt(act, da))$

$getCurrentSt(da) \neq getDesiredOnDASt(act, da) \Rightarrow getCurrentSt(da) =$ RECURSIVE-UPDATE(da, $getCurrentSt(da)$, $getDesiredOnDASt(act, da))$]

- $onUpdate(act, o) \Rightarrow onD(act, o) == False$ ▷ ***[Function Call]***

For the requested activity to *continue*, ongoing-dependent activities may require state change.

In *onUpdate*(a) function, we iterate a loop for each ongoing dependent activity, check if the dependent activities and the dependent of dependent activities have conflicting (multiple) desired states or not (calling RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION function from Algorithm 1) followed by updating their current states by calling the RECURSIVE-UPDATE function in Algorithm 2 (with checking and updating the states of "chain of dependencies"). This *onUpdate* function is called when onD returns $False$ suggesting that not every dependent activity is in desired state.

Example 5. In smart farming, an ongoing activity is $cooling$ the greenhouse requested by the source $farmManager$ (formally stated as $request(farmManager, cooling)$).

- $S = \{farmManager\}$
- $O = \{airCooler, humidifier\}$
- $OP = \{turnOn, turnOff\}$
- $ACT = \{cooling, humidifying\}$
- $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$
- $getObject(cooling) = airCooler$
- $getOperation(cooling, airCooler) = turnOn$
- $getDA(cooling, airCooler) = \{humidifying\}$
- $getCurrentSt(humidifying) = inactive$
- $getDesiredOnDASt(cooling, humidifying) = running$
- $onUpdate(cooling) \Rightarrow onD(cooling, airCooler) == False$

In example 5, the ongoing activity is $cooling$ the environment of a greenhouse using the object $airCooler$. While $cooling$, if the humidity is low the $humidifier$ should be $running$ to continue $cooling$. In that case, $humidifying$ is an ongoing dependent activity for $cooling$. We call the *onUpdate*($cooling$) function and update the current state of $humidifying$ from *inactive* to the $running$ state as the onD function returns $False$. This will ensure that the $cooling$ continues while $humidifying$ is running.

Definition 6. ACAC$_{\mathrm{onD}_3}$: **Post-update model for dependent activities with ongoing-check**

ACAC$_{\mathrm{onD}_3}$ model adds the update procedure for the dependent activities which may need state change when the requested activity is finished, on hold, or revoked, requiring ongoing check. For instance, A is a requested activity and B is a dependent activity which needs to be started while A is running. After A is revoked, B should be stopped immediately. This is a post-update on B based on the decision taken on activity A while running (ongoing check). ACAC$_{\mathrm{onD}_3}$ model is an extension to ACAC$_{\mathrm{onD}_0}$ adding the post-update procedures.

- $getDesiredPostDASt$: $ACT_R \times ACT_D \longrightarrow ST_{DR}$ ▷ ***[mapping a requested activity which has been 'finished', 'revoked', or on 'hold', and a post-dependent activity to a desired state]***
- $postD(act\text{: } ACT,\ o\text{: } O) \longrightarrow \{True,\ False\}$, defined as $\bigwedge_{(da \in getDA(act,o))} getCurrentSt(da) = getDesiredPostDASt(act, da)$
- *postUpdate*(act, o): ▷ ***[Function Definition]***
  $(\forall da \in getDA(act, o))$.
  [RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION(da,$getCurrentSt(da)$, $getDesiredPostDASt(act, da))$

  $getCurrentSt(da) \neq getDesiredPostDASt(act, da) \Rightarrow getCurrentSt(da) =$ RECURSIVE-UPDATE(da, $getCurrentSt(da), getDesiredPostDASt(act, da))$]
- *postUpdate*(act, o) $\Rightarrow postD(act, o) == False$ ▷ ***[Function call]***

In this model, $getDesiredPostDASt$ function provides the desired state of a post-dependent activity. This function takes a requested activity and one of its dependent activity as input and returns a desired state for this dependent activity. $postD$ function checks the current and desired states of the post-dependent activities and returns $True$ or $False$ based on the outcome of the comparison between current and desired states of all post-dependent activities. In *postUpdate* function, the current states of all the dependent activities are updated (to desired states) if they are not in the desired states (i.e., if $postD$ returns $False$). Before updating the states, we check if the dependent activity da or any of its dependent activity has conflicting desired states or not. We call the RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION function from Algorithm 1 (explained in Sect. 4) by passing a post-dependent activity, its current state and its desired state. After that, we call the RECURSIVE-UPDATE function in Algorithm 2 by passing a post-dependent activity, its current state, and its desired state and it returns the desired state after checking and updating the "chain of dependencies".

Example 6. In smart home, $floorCleaning$ was requested by the source $floorWorker$, stated as $request(floorWorker, floorCleaning)$.

- $S = \{floorWorker, sensor\}$
- $O = \{vacuumCleaner, roboticArm\}$
- $OP = \{turnOn, turnOff\}$
- $ACT = \{movingObject, floorCleaning\}$
- $ST = \{inactive, dormant, aborted, running, hold, revoked, finished\}$
- $getObject(floorCleaning) = vacuumCleaner$
- $getOperation(floorCleaning, vacuumCleaner) = turnOn$
- $getDA(floorCleaning, vacuumCleaner) = \{movingObjects\}$
- $getCurrentSt(movingObjects) = running$
- $getDesiredPostDASt(floorCleaning, movingObjects) = inactive$
- $\mathrm{postUpdate}(floorCleaning) \Rightarrow postD(floorCleaning, vacuumCleaner) ==$ False

In Example 6, we assume the activity $floorCleaning$ has been just finished which was running on the object, $vacuumCleaner$. For the continuity of this activity, $movingObjects$ by $roboticArm$ was running. The purpose of $movingObjects$ is done after $floorCleaning$ is finished. Thus, $movingObjects$ needs to be in $inactive$ state as a post-dependent activity. We update the state using the $\mathrm{postUpdate}(floorCleaning)$ function.

4 Challenges of resolving chain of dependencies

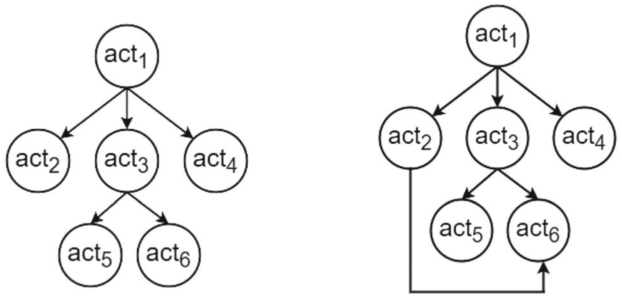
Chain of dependencies refers to "dependencies of dependencies" where one activity relies on another activity for the state transition, which in turn relies on some other activity and these sequence continues until there exists one independent activity which is not dependent on others for its state transition. In large, complex and dynamic environments, the proliferation of activities is inevitable. The dynamic nature of the activities evolving over time and changing the states based on conditions may often pose challenges in managing the policies with manual specifications. Due to the activities having the mutability characteristic, it is essential to keep the dependent activities and chain of dependent activities separate from the specification expressions.

In case of manual specification of the policies, the administrators must ensure that there is no conflicting and deadlock situations created. In this regards, the administrators can use the applications or existing tools to check if the chain of dependencies can form a deadlock by using different combination of current and desired states of the parent and child dependent activities. The administrators also need to check whether an activity is reachable to the desired state while having parallel request processing and non-deterministic order of dependency check and updates. To accommodate the manual specification and address the deployability concern, developing tools and frameworks using algorithms for determining the reachability and existence of deadlocks can help the administrator avoid assigning the conflicting dependencies. In such scenarios, depending on the designer's choice and fulfillment of the system requirements, the dependencies are required to be assigned creating no conflicts. In existing literature, the state of the art works on reachability analysis [28] of different critical components such as attributes can be helpful as resources for the administrators.

The request processing time increases with number of dependencies checked and updated. In real-time environment, the dynamic nature of activities and hundreds or thousands of requests being processed simultaneously can impact the request processing time. In parallel execution of the activities, depending on the priority of activities, few activities may need to wait for other activities to be finished. In addition to that, duration of activities based on the system requirements and other parameters can also impact the request processing time. However, the duration is dependent on the system requirements. In the specification context, the administrators need to avoid the complexities while assigning dependent activities to ensure the system never reach to an unsolvable state due to conflicts and deadlocks. In this section, we discuss the challenges associated with resolving a chain of dependencies. that increase the complexity and reduce flexibility to update the states of dependent activities. In the following subsections, we discuss these challenges.

4.1 Multiple dependency paths: non-deterministic or deterministic?

A requested activity may depend on a single or multiple activities in any phase of its life cycle. Multiple dependency paths (as shown in Fig. 9a and 9b) can lead to increased complexity



(a) Multiple dependency paths without common dependent activities

(b) Multiple dependency paths with a common dependent activity

Fig. 9 Chain of dependencies with multiple dependency paths. The circles represent activities while the arrows indicate that the parent activity (e.g. act_1) depends on the child activity (e.g. act_2) to change its state

in determining the path which the system should take first. On a different note, the order of dependency checks and updates (if required) can raise the question of whether the selection of order should be deterministic or non-deterministic. We define the deterministic and non-deterministic order of dependency check and update and later in this section, we explain which strategy is chosen for the selection of dependency path.

- **Deterministic order of dependency check and update:** In a deterministic order for checking and updating the dependent activity states across multiple dependency paths, we can enforce a very specific selection criteria based on which order of dependency checks among a finite number of dependent activities is determined. In Fig. 9a and 9b, we show two examples of activity dependency chains where act$_1$ has three dependent activities, thus it has multiple dependency paths. In Fig. 9b, act$_6$ is a common dependency for both act$_2$ and act$_3$. For example, we can say that the current state of act$_6$ is "running" and to resolve act$_2$, act$_6$ has to be in the "inactive" state. Moreover, act$_2$ needs act$_6$ to stay in the "inactive" state. On the other hand, to resolve act$_3$, the desired state of act$_6$ is "finished". Conceptually, according to the life cycle of an activity, it goes to an "inactive" state from "finished" state after a certain time if there are no further dependencies (post-dependent activities). We consider such a scenario for act$_6$ in Fig. 9b. If we choose one of these two dependency paths, act$_1$ $\longrightarrow$ act$_3$ $\longrightarrow$ act$_5$ $\longrightarrow$ act$_6$ and act$_1$ $\longrightarrow$ act$_3$ $\longrightarrow$ act$_6$ $\longrightarrow$ act$_5$ starting from act$_1$ followed by act$_3$, act$_6$ will get the state "finished" and it will go to the "inactive" state since there are no other dependencies required to be checked for act$_6$. As a consequence, act$_2$ can be resolved as it can have the act$_6$ in the desired state "inactive" while checking its dependencies. This dependency check and update process is deterministic as we select the starting path comparing two different states of a common dependent activity. This selection also results in the expected outcome by resolving the chain of dependencies. However, this deterministic solution can be difficult to apply to accomplish the ultimate goal where there exists a large number of activities with multiple dependency paths including common dependent activities with different desired states.
- **Non-deterministic order of dependency check and update:** The non-deterministic approach for dependency check and update refers to the strategy where the sequence of activity dependency checks and updates is not fixed as well as unpredictable if an activity has multiple dependency paths. Evaluation of dependencies and update process can vary in the order each time the dependencies are checked for a specific activity. In Fig. 9a and 9b, act$_1$ has three dependent activities, thus it has multiple dependency paths. In a non-deterministic selection of dependency path, the criteria to select the order of checking and updating the states of dependent activities (if required) is not predefined by the system. It can be randomly chosen and the external system does not have access to know the selection process.

In Fig. 9a and 9b, we show six activities in the circles named act$_1$, act$_2$, act$_3$, act$_4$, act$_5$, and act$_6$. act$_1$ is the requested activity, thus we can refer to it as the root of the dependency chain. Both the (a) and (b) in Fig. 9 include act$_2$, act$_3$, and act$_4$ as dependent activities of act$_1$. For instance, we can think of these three activities as pre-dependent activities of act$_1$ which means we need these three activities in their respective desired states before starting act$_1$. The difference between (a) and (b) in Fig. 9 is the parent activities of act$_6$. In Fig. 9a, act$_3$ depends on act$_6$ along with act$_5$ whereas in Fig. 9b both the act$_2$ and act$_3$ depend on act$_6$ for their state change into the respective desired states. In the first Fig. 9a, there is no common dependency which means every dependent activity has only one parent activity in the dependency chain. On the contrary, in 9b, act$_6$ is a common dependent activity for both act$_2$ and act$_3$. For the first instance in 9a, there is no complex situation while resolving the chain of dependencies since all the dependent activities can change their current state to the desired state (if required) for their corresponding parent activities. Thus, the order of evaluating the dependencies and the update process does not matter in this scenario. Therefore, whether we choose deterministic or non-deterministic approach for dependency checks and update for dependency chains does not matter where there are no common dependencies between two or more parent activities.

In Fig. 9b, in the dependency chain of the requested activity (act$_1$), act$_2$, and act$_3$, both depend on act$_6$ in order to change their current state to the respective desired states. In this instance, there can be one of the two possible cases; requiring the same desired state of act$_6$ for both of these activities (act$_2$ and act$_3$) or requiring different desired states of act$_6$ for each activity. There does not exist any conflict if act$_6$ requires to be in the same desired state in order to change states (to their desired ones) of act$_2$ and act$_3$. However, conflict will arise when act$_2$ and act$_3$ require two different desired states for act$_6$. We refer to these different desired states as "conflicting desired states". In scenarios where a dependent activity has conflicting desired states, we may choose deterministic order of dependency check and update that can provide an ultimate result where the root activity (act$_1$ in Fig. 9b) can certainly make its transition to the desired state. However, we cannot guarantee the expected outcome for the root activity of the dependency chain even if we take a deterministic solution. For example, we can compare the conflicting desired states of the common dependent activity (act_6) and take the most preceding state among those different desired states so that the common dependent activity can get a scope to transition to the next desired states. We need to backtrack to determine the order of the paths from the root activity to the common dependent activity with the most preceding desired state. However, we may not be able to get the desired outcome in this deterministic solution if the common dependent activity (act_6) needs to remain in a specific state (e.g. "inactive") to change the current state of one the parent activities (e.g. act_2). Moreover, act_2 needs to hold the specific desired state to change its state first, and then act_1 (root activity) can change its state. On the other hand, act_3 needs to change the state of act_6 to "running" state in order to change the state of its parent activity (root activity "act_1"). In this case, act_2 will be able to hold (which we also refer to "lock") act_6 in the desired state of "inactive", thus act_3 cannot change it to the "running" state. This is a policy conflict that cannot be solved either we choose a deterministic or non-deterministic approach and it certainly cannot provide any desired outcome for the root activity (act_1). This is a policy design issue that should be handled while designing the policy and must be avoided to resolve a chain of dependency with multiple dependency paths problems.

When deciding about the deterministic approach to resolve the chain of dependencies, it becomes more complex when there are multiple levels of dependencies including activities with multiple desired states. Finding the specific order for every single activity chain is not flexible and scalable. Therefore, the system may choose the order of dependency check and update and we can leave it as a non-deterministic approach. However, choosing a non-deterministic order may sometimes lead to a race condition state. In the following section, we will address this problem and provide a solution for it.

4.1.1 Race condition problem with non-deterministic order of dependency check and updates with multiple desired states

In non-deterministic execution order, we need to make sure that the state of a common dependent activity with conflicting desired states cannot be overwritten or updated when its parent activity (in the selected path from multiple dependency paths using non-deterministic order) needs the common dependent activity in a specific state. Since an activity is a long continuous event, there may exist a scenario where the dependent activity fulfills the requirement and later, it can change the state according to the system context and design. Here, the race condition refers to "racing" to modify the common dependent activity's state by multiple parent activities. We need to make sure the system does not allow a parent activity to change the common dependent activity's state while another parent activity wants it to stay in another conflicting state. This race condition formulates a problem of how the system can handle the situation where a parent activity holds a dependent activity with conflicting (multiple) desired states in a specific state for a certain duration and this state cannot be overwritten by any other activity at the same time. We propose a solution using the following steps.

- Initially, we check whether there exist conflicting desired states (multiple) for the dependent activities in a chain of dependencies. We store this information for future usage (referred to as Algorithm 1).
- We introduce a recursive update process for dependent activities (in Algorithm 2) where it completes the updates if the dependent activities fulfill the requirements of desired states. If there are conflicting desired states for a dependent activity, we use the locking mechanism (Algorithm 3) for the dependent activity. The lock remains until the parent activity's purpose is served.

Algorithm 1 is utilized to determine whether a dependent activity possesses conflicting desired states, where multiple parent activities require different desired states for the dependent activity. This algorithm consists of two functions: RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION(da, $da\_current\_state$, $da\_desired\_state$) and DETECT-CONFLICTING-DESIRED-STATE ($da, da\_desired\_state$). In the RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION(da, $da\_current\_state$, $da\_desired\_state$) function, da represents the dependent activity for which conflicting desired states are detected, with $da\_current\_state$ representing its current state and $da\_desired\_state$ denoting the desired state of the dependent activity. The function DETECT-CONFLICTING-DESIRED-STATE($da, da\_desired\_state$) is employed to identify conflicting desired states. It takes the dependent activity da and the currently examined desired state ($da\_desired\_state$) as inputs. In this function, line 1 checks if a desired state is already assigned to da using the *assignedDesiredState*(da) function. If the function returns an empty set (∅), we assign the currently examined desired state, $da\_desired\_state$, as the result. At this stage, as no other desired state has been checked for da, we can infer that no conflicting desired state exists for da and assign "FALSE" as the result of *hasConflictingDesiredState*(da). However, if there is a difference between the currently examined desired state $da\_desired\_state$ and the assigned Desired State (*assignedDesiredState*(da)), we conclude that the dependent activity da possesses conflicting desired states and assign "TRUE" as the result of *hasConflictingDesiredState*(da). After executing line 1 (calling DETECT-CONFLICTING-DESIRED-STATE(da, $da\_desired\_state$)), we identify

Algorithm 1 Detecting Conflicting Desired States of Dependent Activities

```
RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION($da$, $da\_current\_st$, $da\_desired\_st$):
Description: detects conflicting desired states for a chain of dependent activities.
Input: $da$: a dependent activity

$da\_current\_st$: the current state of the dependent activity $da$

$da\_desired\_st$: the desired state of the dependent activity, $da$.
1: DETECT-CONFLICTING-DESIRED-STATE($da$, $da\_desired\_st$)
2: $DoDA = getDoDA(da, da\_current\_st, da\_desired\_st)$
3: if ($DoDA \neq \emptyset$)
4: then
5: for (each $doda \in DoDA$) do
6:       RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION($doda$, $getCurrentSt(doda)$,
7:         $getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda)$)
8: end for
9: end if
DETECT-CONFLICTING-DESIRED-STATE($da$, $da\_desired\_st$):
Description: detects conflicting desired states and stores the information for a dependent activity.
Input: $da$: a dependent activity
$da\_desired\_st$: the desired state of the dependent activity $da$
1: if ($assignedDesiredSt(da) == \emptyset$)
2: then $assignedDesiredSt(da) = da\_desired\_st$

3: $hasConflictingDesiredSt(da)$ = FALSE
4: else if $da\_desired\_st \neq assignedDesiredSt(da)$
5: then $hasConflictingDesiredSt(da)$ = TRUE
6: end if
```

conflicting desired states for "dependencies of dependencies." In line 2, we obtain the set of dependent of dependent activities, $DoDA$, for the dependent activity da. Line 3 checks if this set is empty. If $DoDA$ is not empty, we recursively call the RECURSIVE-CHECK-OF-DEPENDENCIES-WITH-CONFLICT-DETECTION function for each "dependent of dependent" activity to detect conflicting desired states for these activities.

Algorithm 2 is implemented to handle the recursive process of checking and updating the states of dependent activities within a dependency chain. We choose a recursive structure for this algorithm to ensure that we address the "dependencies of dependencies" before updating the state of a dependent activity. The function RECURSIVE-UPDATE(da, $da\_current\_st$, $da\_desired\_st$) is defined, where da represents a dependent activity, $da\_current\_st$ denotes its current state, and $da\_desired\_st$ denotes the desired state for da. In line 1, we obtain the set of dependent activities ($DoDA$) that are required for da to transition from $da\_current\_st$ to $da\_desired\_st$. If $DoDA$ is empty, it implies that there are no dependencies that need to be checked for this specific state transition of da. Line 2 verifies whether $DoDA$ is empty or not. If the condition is true, we return $da\_desired\_st$ from the function (line 3). Otherwise (line 5), we proceed to explore each activity ($doda$) in $DoDA$. Within lines 6-7, we check whether the current state and desired state of each $doda$ in $DoDA$ are not the same and whether $doda$ has any conflicting state. This information has already been stored using Algorithm 1 for all activities in the dependency chain. If the condition in lines 6-7 is met, we update the state of $doda$ by calling the RECURSIVE-UPDATE function, providing $doda$, its current state, and desired state as parameters (lines 8-10). This recursive call is necessary to check if $doda$ has any further dependent activities and to compare their current and desired states before returning the desired state. We include an additional check to verify if the current and desired states of $doda$ are not equal and if $doda$ has any conflicting states (lines 11-12). If these conditions are satisfied, we call ACQUIRE-LOCK function defined in Algorithm 3 (line 14-15). Algorithm 3 will check if the activity, $doda$ is locked or unlocked. Then we update the current state of this activity by calling the function RECURSIVE-UPDATE (line 16-17).

Algorithm 3 is inspired by the Binary Semaphore [29] or Mutex Lock mechanisms in operating systems. The binary Semaphore mechanism is used to synchronize between two values, 0 and 1, and allows only a single unit to the critical section (to get access to shared resources). We use a similar locking mechanism using a function named "$getBinarySemaphoreValue(doda)$" where $doda$ is a dependent of dependent activity and the value returned from this function is 0 or 1. When the value returned from

Algorithm 2 Recursive Update of States for Chain of Dependent Activities

```
RECURSIVE-UPDATE(da, da_current_st, da_desired_st):
Description: Recursively updates the states of dependent activities while exploring the dependencies of dependencies and updating them first.
Input: $da$: a dependent activity

$da\_current\_st$: the current state of the dependent activity $da$

$da\_desired\_st$: the desired state of the dependent activity, $da$.
Output: Returns a desired state for the dependent activity, $da$.
DoDA = $getDoDA(da, da\_current\_st, da\_desired\_st)$
if (DoDA == ∅)
then return da_desired_st;
else
for (each $doda \in DoDA$) do
    if $(getCurrentSt(doda) \neq getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda)$
    ∧ hasConflictingDesiredSt(doda) == FALSE)
    then
        $getCurrentSt(doda)$ = RECURSIVE_UPDATE(doda, $getCurrentSt(doda)$,
          $getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda))$
     else if $(getCurrentSt(doda) \neq getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda)$
            ∧ hasConflictingDesiredSt(doda) == TRUE)
     then
          ACQUIRE-LOCK($da, da\_current\_st, da\_desired\_st, doda, getBinarySemaphoreValue(doda)$,
          $getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda))$
          $getCurrentSt(doda)$ = RECURSIVE_UPDATE(doda, $getCurrentSt(doda)$,
          $getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda))$        ▷ RELEASE-LOCK(doda) will be called when
  the purpose of locking doda is done for da
      end if
end for
return da_desired_st
end if
```

"$getBinarySemaphoreValue(doda)$" is 1, this indicates that $doda$ is currently not locked by a parent activity. When the value returned from "$getBinarySemaphoreValue$ $(doda)$" is 0, this indicates $doda$ is currently locked by a parent activity. Therefore, it cannot change its current state to fulfill the requirement of any other parent activity. In the function ACQUIRE-LOCK(da, $da\_current\_st$, $da\_desired\_st, doda, getBinarySemaphoreValue(doda)$, $getDesiredDoDASt(da$,$da\_current\_st$,$da\_desired\_st$, $doda)$), da is the dependent activity which is currently trying to change the current state of $doda$ to the desired state in order to transition from $da\_current\_st$ to $da\_desired\_st$. In this algorithm, we check if $doda$ is currently locked ($getBinarySemaphoreValue(doda)$ == 0) or unlocked ($getBinarySemaphoreValue(doda)$== 1). If it is unlocked, we change the value of $getBinarySemaphoreValue(doda)$ to 0 which indicates it is locked by da (line 1-2). If $doda$ is locked by some other activity (line 3), da must wait for $doda$ to be unlocked until we get the value 1 from $getBinarySemaphoreValue(doda)$ (line 5-7). $waitFor(doda)$ indicates, the parent activity da will wait for $doda$ to be unlocked. Once the previous parent activity releases the lock using RELEASE-LOCK($doda$) and changes the value of $getBinarySemaphoreValue(doda)$ to 1, the currently waiting dependent activity da again calls the ACQUIRE-LOCK function and updates the value of $getBinarySemaphoreValue(doda)$ (locks $doda$) (line 8-9).

4.2 Circular dependencies and deadlock

In our proposed ACAC$_D$ model, there can be circular dependencies that create a deadlock situation. In a circular set of dependencies, the chain of dependencies is created in a circular fashion (shown in Fig. 10). In this figure, the dependency path is act$_1$ $\longrightarrow$ act$_2$ $\longrightarrow$ act$_3$ $\longrightarrow$ act$_1$. In this circular set of activities, act$_1$ depends on act$_2$ to find act$_2$ in the desired state "finished". act$_2$ depends on act$_3$ and requires act$_3$ to be "finished". act$_3$ requires act$_1$ to be "running" before it goes to "finished" state. This circular wait for each activity is in a deadlock and no activity ultimately gets their desired state. There are certain ways to handle this type of deadlock situations. We discuss the deadlock handling techniques in the next subsection.

Algorithm 3 Locking Mechanisms for Activities with Conflicting Desired States

```
ACQUIRE-LOCK($da, da\_current\_st, da\_desired\_st, doda, getBinarySemaphoreValue(doda)$
$getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda)$):
Description: Lock a dependent of dependent activity if it is unlocked and wait for the release of lock if it is locked.
Input: $da$: a dependent activity,
$da\_current\_st$: the current state of the dependent activity $da$,
$da\_desired\_st$: the desired state of the dependent activity, $da$.
$doda$: a dependent of dependent activity.
$getBinarySemaphoreValue(doda)$: the binary semaphore value of $doda$ which can be 0 or 1 in turn.
$getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda)$: the desired state of $doda$ corresponding to the
parent activity $da$'s state transition from $da\_current\_st$ to $da\_desired\_st$.
if($getBinarySemaphoreValue(doda)$==1)
then $getBinarySemaphoreValue(doda)$ = 0;
else if ($getBinarySemaphoreValue(doda)$==0)
then
while ($getBinarySemaphoreValue(doda)$==0) do
    waitFor(doda)
end while
ACQUIRE-LOCK($da, da\_current\_st, da\_desired\_st, doda, getBinarySemaphoreValue(doda)$,
$getDesiredDoDASt(da, da\_current\_st, da\_desired\_st, doda)$)
end if

RELEASE-LOCK($doda$):
Description: releases the lock for a dependent of dependent activity.
Input: $doda$: a dependent activity
$getBinarySemaphoreValue(doda)$ = 1;
```

4.2.1 Deadlock detection and solutions

The system may fall into a deadlock if the system administrator fails to prevent it from assigning the dependencies that create a circle. The deadlock due to a circular set of dependent activities can be detected before the update process starts. We can detect this deadlock with a typical Depth First Search (DFS) algorithm. This kind of deadlock situation needs to be carefully analyzed by the system designer. Upon identifying a circular dependency in the system, the administrator plays a crucial role in breaking the cycle within the dependency chain. It becomes imperative for the administrator to thoroughly examine the activities involved in the cycle and pinpoint a low-priority activity that can be strategically removed. If the administrator successfully accomplishes this task, the deadlock can be effectively eliminated, ensuring the system's smooth operation without violating the safety rules. Therefore, careful analysis and decision-making by the administrator are instrumental in resolving such deadlock scenarios, ultimately optimizing the system's performance and preventing potential disruptions. Maximum timeout mechanisms can also be applied for a requested activity where the request is denied after a certain period of time. Deadlock detection and recovery are challenging for a chain of dependent activities since this is a design choice and policy engineering problem. Deadlock prevention is a more suitable deadlock handling method for a chain of dependent activities in such smart systems referred to by our proposed ACAC model.

4.3 Combination of ACAC$_D$ sub-models while resolving chain of dependencies

As described in Sect. 3, different sub-models, denoted as ACAC$_D$ sub-models, support the mutability of activities at various stages of their life cycle. Throughout the paper, we discuss a set of several states $\{inactive, dormant, aborted, running, hold, revoked, finished\}$ that a requested activity can pass through from its initiation to completion. To fulfill the request for an activity, the states of dependent activ-

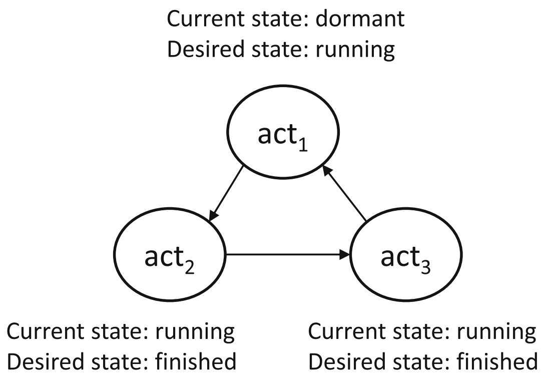


Fig. 10 Circular dependencies of activities. The circles denote activities and the arrows denote that the parent activity depends on the child activity

ities are examined and updated, if necessary, to enable the transition of the requested activity from one state to another. For instance, when transitioning from the *inactive* to *running* state, the $\mathrm{ACAC}_{\mathrm{preD}_1}$ model is employed to verify and modify the states of the dependent activities required to initiate the requested activity. During the *running* state of the requested activity, the $\mathrm{ACAC}_{\mathrm{onD}_2}$ model is utilized for conducting ongoing checks and updates. Consequently, it can be inferred that a combination of different $\mathrm{ACAC}_{\mathrm{D}}$ sub-models supports the successful execution of a requested activity from start to finish.

A "chain of dependent activities," also known as "dependencies of dependencies", requires checks for current and desired states of the dependent activities and updates to the current states to accommodate state transitions of parent activities. As illustrated in Fig. 7 in Sect. 3, consider the example where the activity "Water Spraying" requires "Nitrogen Spraying" to be in an *inactive* state while it is currently in a *running* state. Similarly, the state transition of "Mixing Sawdust to soil" from *running* to *inactive* is required. Suppose "Nitrogen Spraying" first transitions to a *finished* state before reaching the *inactive* state. Since it has no post-dependent activities, it will go to *inactive* state. In this scenario, "Mixing Sawdust to soil" is checked and updated when "Nitrogen Spraying" is ongoing, and a decision is made to finish the ongoing activity. Based on the definitions of our $\mathrm{ACAC}_{\mathrm{D}}$ models, $\mathrm{ACAC}_{\mathrm{onD}_2}$ model can be utilized to check and update the "Mixing Sawdust to soil" activity, while $\mathrm{ACAC}_{\mathrm{onD}_3}$ model ensures there are no post-dependent activities. Hence, combination of $\mathrm{ACAC}_{\mathrm{D}}$ sub-models proves to be an effective approach for resolving the chain of dependencies.

5 Prototype implementation

In this section, we present a prototype implementation of a combination of $\mathrm{ACAC}_{\mathrm{D}}$ sub-models in a smart farming use case (as shown in Fig. 1). The code is written in Python 3 using PyCharm on Hp Envy x360 convertible with Intel core i7 processor and 12 GB of RAM. The implementation shows the need for different $\mathrm{ACAC}_{\mathrm{D}}$ sub-models to incorporate the dependencies (D) in the activity request decision. In a fully deployed ACAC model, all four decision parameters (Authorizations (A), Obligations (B), Conditions (C), and Dependencies (D)) will be considered. However, since our paper focuses on the $\mathrm{ACAC}_{\mathrm{D}}$ models for activity dependencies, we evaluate these sub-models, assuming other parameters are satisfied. We have simulated the devices and activities in the system; however, this does not undermine the plausibility, use, and advantage of our proposed $\mathrm{ACAC}_{\mathrm{D}}$ model, as further elaborated in the following discussion.

5.1 Description of the use case

A smart farming ecosystem consists of connected smart devices that perform multiple activities concurrently. There are inter-dependencies among activities that may constrain the execution of other activities. This requires checking and updating the states of dependent activities to make any activity request decision. In Table 4, we include four activity requests in the first column. Each request has two parameters; the first and second parameter indicates the requesting source and the requested activity, respectively. The second, third, and fourth columns include *pre-*, *ongoing*, and *post-* dependent activities, respectively. We also mention the desired states (such as *running* or *inactive*) of dependent activities after the colon ':'. Since the current states of the activities depend on the real-time system context, these are not specified. Further, we implement an activity request with its chain of dependencies. In Table 5, we include the dependencies of dependencies corresponding to the first request shown in Table 4. The first column indicates the name of the dependent activities, the second and third columns indicate its current and desired states respectively. The fourth column includes the dependent of dependent activities corresponding to the transition from the current state to the desired state of the parent dependent activities mentioned in the first column. We also mention the corresponding desired states of the dependent of dependent activities followed by a colon ':'.

5.2 Use case implementation

To implement the use case and satisfy the activity requests in Table 4, we configure five JSON files as follows, `request.json` (have the activity requests with a source and requested activity), `activity.json` (includes the

Table 4 Description of activity requests and dependencies

Requests	Pre-dependent activities	Ongoing-dependent activities	Post-dependent activities
request(fieldWorker, sprayingWeedKiller)	– *mixingAMS* : *finished* – *thermalImaging* : *running*	– *waterSpray* : *inactive* – *thermalImaging* : *running*	– *waterSpray* : *inactive* – *pullingWeedsUp* : *running* – *weedScanning* : *running*
request(farmer, sowingSeeds)	– *fieldPloughing* : *inactive*	– *pesticideSpray* : *running* – *thermalImaging* : *running* – *airCooling* : *running*	N/A
request(farmManager, fieldPloughing)	– *stakingBoundaries* : *finished* – *mixingWaterAbsorbingMaterial* : *running*	– *waterSpray* : *inactive* – *thermalImaging* : *running*	– *sprayingWeedKiller* : *running* – *sowingSeeds* : *running* – *pesticideSpray* : *running*
request(fieldOwner, coolingGreenhouse)	– *thermalImaging* : *running*	– *humidifying* : *running*	N/A

Table 5 Chain of dependencies for the dependent activities in the first request from Table 4

Dependent activity	Current state	Desired state	Dependent of dependent activity: desired State
mixingAMS	*running*	*finished*	*Dash mixingVinegar*: *running*
pullingWeedsUp	*inactive*	*running*	*Dash pesticideSpray*: *running*
mixingVinegar	*inactive*	*running*	*Dash mixingWater*: *running*

current states of all the activities), `object.json` (holds the objects the activities can be performed on), `operation.json` (contains the operation to perform an activity on a specific object for all the activities), and `activityDependencies.json` (provides the sets of pre-, ongoing- and post-dependent activities with their desires states and against particular object for each requested activity). `activity.json` file is dynamically updated according to the changes made in the current states of dependent activities. Further, we configure another JSON file named `dependenciesOfdependencies` to implement this use case with chain of dependencies where pre-, ongoing and post-dependent activities (for a particular activity request) also have dependent activities to make their transition from the current state to a desired states while satisfying the requested activity's requirements.

As mentioned in Table 4, for the *request(fieldWorker, sprayingWeedKiller)*, we have all three of pre-, ongoing and post-dependent activities with desired states. The current states we get from our `activity.json` file is compared to the desired states. For this pre-dependency check, our implementation procedure supports ACAC$_{\mathrm{preD}_0}$ sub-model. The activity *mixingAMS* (mixingAMS is the short form of mixingAmmoniumSulfate) is initially in *running* state which needs to update its state to the desired state *finished*. This update occurs in the enforcement point as supported by the ACAC$_{\mathrm{preD}_1}$ sub-model. In a similar way, the ongoing dependent activities are checked, and the current state of *waterSpray* is updated from *running* to *inactive*. In this ongoing check, the sub-models ACAC$_{\mathrm{onD}_0}$ (for checking the states of ongoing dependent activities) and ACAC$_{\mathrm{onD}_2}$ (for updating the current states of the ongoing-dependent activities) are applicable. In post-check, a post-dependent activity, *pullingWeedsUp* needs to change its state (from *inactive* to *running*) where the sub-model ACAC$_{\mathrm{onD}_3}$ fits the best. In summary, this use case implementation shows the combination of ACAC$_{\mathrm{preD}_0}$, ACAC$_{\mathrm{preD}_1}$ ACAC$_{\mathrm{onD}_0}$, ACAC$_{\mathrm{onD}_2}$, and ACAC$_{\mathrm{onD}_3}$ for satisfying the *request(fieldWorker, sprayingWeedKiller)*. Similarly, for the other requests, the same procedure repeats for pre-, ongoing, post-check and thus, reflecting the applicability of our proposed ACAC$_{\mathrm{D}}$ sub-models.

Table 6 Execution time for pre-, ongoing, and post-check. **NDC** denotes Number of Dependent Activities Checked and **NDU** denotes Number of Dependent Activities Updated against total number of requests

	Pre-Check			Ongoing-Check			Post-Check		
Number of Requests	NDC	NDU	Time	NDC	NDU	Time	NDC	NDU	Time
10	20	10	38.84	30	10	42.83	20	20	15.21
20	30	10	53.33	60	30	57.64	20	20	22.09
30	50	0	101.33	80	0	87.24	50	10	25.44
40	60	10	110.16	90	10	124.3	50	30	60.76

To implement this use case with a chain of dependencies, we consider the first request from Table 4 which is *request (fieldWorker, sprayingWeedKiller)*. As mentioned

Table 7 Execution time for pre-, ongoing, and post-check with resolving chain of Dependencies. **NDC** denotes Number of Dependent Activities Checked and **NDU** denotes Number of Dependent Activities Updated against total number of requests

	Pre-Check			Ongoing-Check			Post-Check		
Number of Requests	NDC	NDU	Time	NDC	NDU	Time	NDC	NDU	Time
10	40	30	45.89	30	30	32.35	30	10	40.14
20	80	60	79.06	60	60	59.93	80	60	71.56
30	120	90	116.16	90	90	84.56	120	90	99.97
40	160	120	194.32	120	120	94.81	160	120	138.1

in Table 5, we have the dependent of dependent activities corresponding to the transition (from current state to desired state) of parent dependent activities. To implement this request with the chain of dependencies, we configure a JSON file `dependenciesOfdependencies.json`. In pre-, ongoing, and post-check of *request(fieldWorker, sprayingWeedKiller)*, the dependencies of dependencies are checked and the updates are resolved recursively where all the required updates are performed for the dependencies before its parent activity transitions to the desired state. For instance, before updating the state of *pullingWeedsUp* from *inactive* to *running* in the post-check, we check the dependencies of dependencies and update their states accordingly if required, (*pesticideSpray* updates its state from *inactive* to *running* for the particular required transition of *pullingWeedsUp*). The dependent activities which are not mentioned in Table 7 do not have other dependent activities (DoD). In general, the dependencies that are checked when a parent activity's current state is *inactive* and needs to transition to a *running* state, are called pre-dependent activities. Similarly, ongoing dependencies are checked while the parent activity's current state is *running* and needs the transition to any succeeding state (such as *finished*, or *hold*). Ongoing dependencies are also checked regularly to see whether the execution could continue or be revoked. The post-dependent activities are checked when parent activity's required state transitions are *finished* or *revoked* to *inactive*, or *hold* to *running*, *finished* or, *revoked*.

The sequence of the implementation process is shown in Fig. 11. We have three phases (shown in different colors) of checking and updating the dependent activity states while satisfying the requests, referred as pre-check, ongoing check, and post-check. When a source requests an activity, it is checked at the policy decision and enforcement point, the suitable object and operation are selected (mentioned as *getObject(activity) and getOperation(activity, object)*) from the `object` and `operation` finder modules, respectively, which check the `object.json` and the `operation.json` files. In pre-check phase, the activity dependency module provides the pre-dependent activities using the `activityDependencies.json`. In the policy decision and enforcement point, for each pre-dependent activity, current and desired states are checked and updated (if required and depending on mutability).

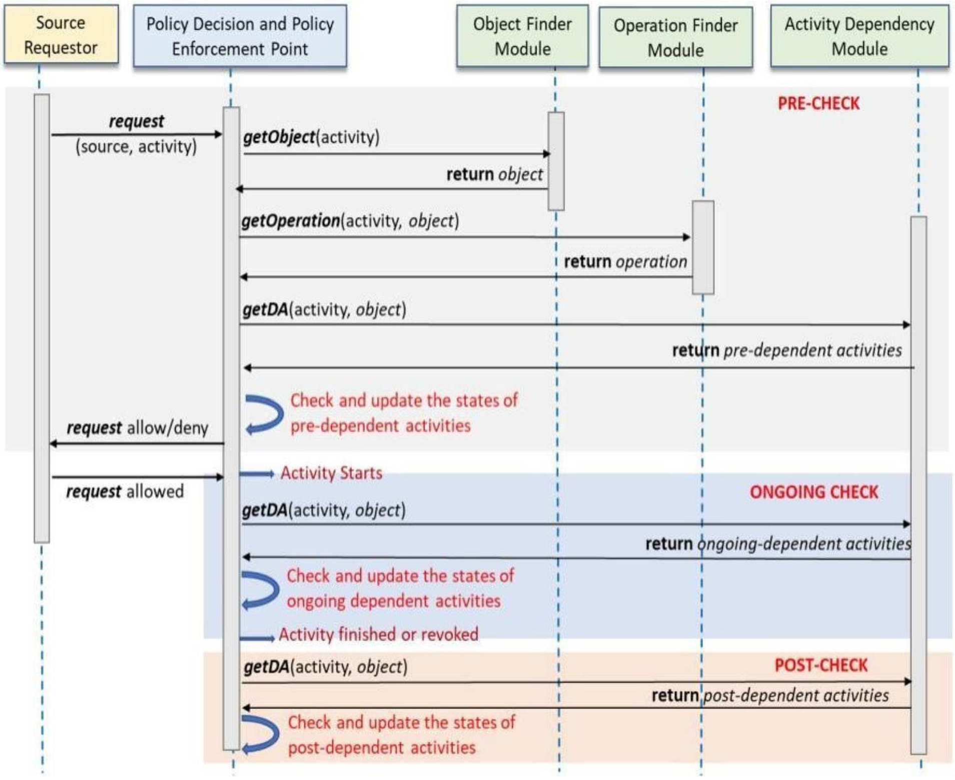
In our implementation without dependencies of dependencies (Table 4), the dependent activities directly update their current states without checking further dependencies. On the other side, the implementation with dependencies of dependencies (first request from Table 4 and chain of dependencies of this requested activity in Table 5), in RECURSIVE-UPDATE function call (mentioned as *RECURSIVE-UPDATE(dependent activity, current state of dependent activity, desired state of dependent activity)*), further dependency check (using `dependenciesOfdependencies.json` file) and the recursive update take place. The request is allowed or denied based on the fulfillment of the dependencies. The activity starts to run at this point. In ongoing phase, the ongoing dependent activity states are checked and updated. We assume the requested activity is finished after resolving the ongoing dependencies. Similarly, post-dependent activity states are checked and updated in the post-check after the activity is revoked or finished. The requested activity changes its current state (from *finished* or *revoked* to *inactive*) at this point.

5.3 Performance evaluation

We evaluated the implementation of our proposed $\mathrm{ACAC_D}$ model in different processing stages (pre-, ongoing, and post-check). We evaluate our prototype for the four activity requests stated in our use case by sending each activity request ten times simultaneously (assuming ten different sources request for ten different activities) and adding new requests in the same proportion.

Table 6 shows the execution time (in milliseconds) against the total number of requests for pre-check, ongoing check and post-check respectively. The first column indicates the number of requests. The first and second sub-columns in each of the second, third and fourth columns indicate the number of dependent activities checked (NDC) and the number of dependent activities updated (NDU) for the number of requests indicated in the first column, respectively in pre-check, ongoing-check and post-check. It must be noted that, in pre-check, the current state of a requested activity is updated from *inactive* to *running* if it is allowed after checking and updating the current states of the pre-dependent activities. In this case, we start the timer when the request is made and calculate the execution time until it updates the current state if the activity is allowed. In ongoing-check, after checking and updating the ongoing dependent activities, we assume the requested activity is finished and, thus, update its current state from *running* to *finished*. The execution time is then evaluated for the duration of the dependency checking and updating the ongoing dependent activity states (if required) and changing the current state of the ongoing

Fig. 11 Sequence diagram for ACAC$_D$ Implementation


Source Requestor
Policy Decision and Policy Enforcement Point
Object Finder Module
Operation Finder Module
Activity Dependency Module
PRE-CHECK
request (source, activity)
getObject(activity)
return object
getOperation(activity, object)
return operation
getDA(activity, object)
return pre-dependent activities
Check and update the states of pre-dependent activities
request allow/deny
Activity Starts
request allowed
ONGOING CHECK
getDA(activity, object)
return ongoing-dependent activities
Check and update the states of ongoing dependent activities
Activity finished or revoked
POST-CHECK
getDA(activity, object)
return post-dependent activities
Check and update the states of post-dependent activities

requested activity from *running* to *finished*. The execution time for post-check indicates the duration of checking and updating (if required) the post-dependent activities. In our implementation (without chain of dependencies), the execution time of pre- and ongoing checks is more than the execution time of post-check since they perform more dependent activities' states update. It should be noted that the number of updates on dependent activities may reduce as more activities are requested since it is possible that the earlier activity requests have already updated the states, and no more state change is needed for future requests.

Figure 12 compares the execution time against the number of requests considered for pre-, ongoing, and post-check (indicated by blue, red, and green lines, respectively). The figure shows that the execution time increases with the increase in the number of dependent activities checked and updated. We observe that the maximum calculated time is for the forty simultaneous activity requests in the ongoing check case. Since in our use case, this scenario has the maximum number of dependent activities checked along with updates to the current states of requested activities (assuming activities finished their execution). Clearly, the number of dependencies for a particular requested activity and the number of state updates impact the processing time of an activity request.

In implementing the request processing of the chain of dependencies for the first request in Table 4, we evaluate the performance by sending the same request 10, 20, 30, and 40 times. Each time, the dependencies of dependencies are checked, and their states are updated if required. This process is done in a recursive manner to ensure that dependencies are resolved before the parent activity's state changes. Table 7 shows the execution time (in milliseconds) against the total number of requests (in a similar way as done in Table 6). Figure 13 compares the execution time against the number of requests, similarly shown in Fig. 12. Here, we observe that execution time increases with the increase in total number of dependency checks and dependency updates. Since NDC and NDU are the highest in number in pre-check, the execution time is also high in pre-check.

We understand that the processing time will increase with hundreds of devices and activities running simultaneously in a real-world environment. However, in this implementation, we reflect on the plausibility and applicability of considering dependencies as a critical component to support activity control in smart systems.

6 Related work

With the advancement of technologies and growth in IoT devices, the possibility of violation of security mechanisms increases. Various research works, including [30, 31] investigate security and privacy issues existing in smart and connected systems. Yao et al. describe security and privacy

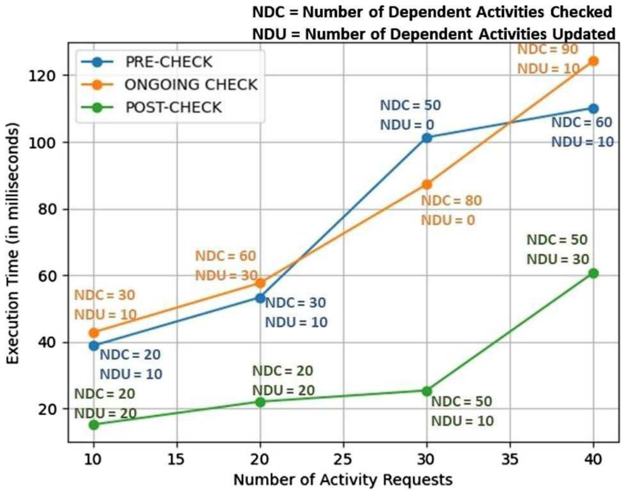


Fig. 12 Performance evaluation of the implementation without chain of dependencies by comparing the execution time (in y-axis) against the number of requests (in x-axis) considered for pre-, ongoing, and post-check (indicated by blue, red, and green lines, respectively). **NDC** denotes the number of dependent activities checked and **NDU** denotes the number of dependent activities updated

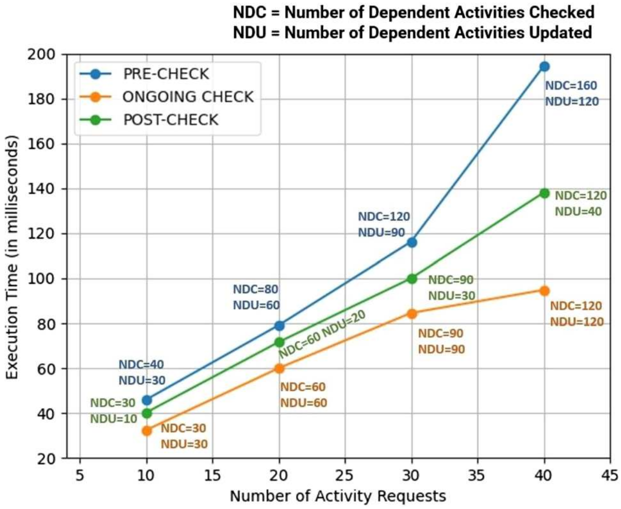


Fig. 13 Performance evaluation of the implementation with chain of dependencies by comparing the execution time (in y-axis) against the number of requests (in x-axis) considered for pre-, ongoing, and post-check (indicated by blue, red, and green lines, respectively). **NDC** denotes the number of dependent activities checked and **NDU** denotes the number of dependent activities updated

challenges in different working stages of physical objects in IoT [30]. Access control solutions have also been proposed for the smart and automated systems, including fine-grained attribute-based access control (ABAC) [9, 26, 32–34].

An attribute-based access control solution for industrial IoT proposed by Bhatt et al. [8] implement their model in Amazon Web Services IoT. Ameer et al. proposed ABAC for secured smart home IoT [32]. These authors introduce and compare HABAC$_\alpha$ (an attribute-based access control model for smart-home IoT) with the EGRBAC (extended generalized role-based access control). The configurations for the role-based approach are mapped with the attribute-based models using user/session, environment, device, operation, and more than one type of attribute. Recently, Sikder et al. introduced a mechanism KRATOS+ for multi-user multi-device access management in Smart home system [35]. They implement the idea using four components; user interaction module, backend server, policy manager, and policy execution module. In the user interaction module, the priority management data and device policies are collected. This work presents the policy negotiation algorithm and maps the policy to a rule. However, this work is very specific to multi-user shared device environments such as smart homes.

Relationship-based access control (ReBAC) models [36–38] have been used to incorporate relations between entities as an access parameter. Multilevel relationships are expressed using ABAC models according to this research. Bayreuther and others recently proposed a task planning for a humanoid robot [39], which converges to the activity-centric access control [14, 15] and usage control [22] showing a structure to incorporate policies, objects, modeling framework, architecture and enforcement of the access control system. The authors discuss a decentralized architecture for the policies and task modeling and gain the enforcement of activity-centric and usage-based access control for robot task planning. However, this work lacks the idea of leveraging both models, which is critical for a smart environment. Mawla et al. proposed [15] a framework for the activity-centric access control model components to check an activity request. These components fit well to address scenarios that consider activity dependencies and other decision factors.

Furthermore, several blockchain-based access control solutions are proposed by researchers [40–43]. Tan et al. propose a blockchain-based access control for the Green Internet of Things (GIoT) for the purpose of saving energy. In this approach, the permission data and identity data are immutable. If we compare this solution to our approach, ACAC is more suitable for scenarios with a large number of devices, a dynamic environment, and supporting dependencies among different activities in smart and collaborative systems. A deep learning-based access control (DLBAC) is proposed by Nobi et al. [44] addresses major limitations of classical access control approaches such as RBAC and ABAC models. This work is significant since it fully automates access control using deep learning. However, it has not been used for large-scale, complex, and dynamic environments due to a lack of accurate access control decisions.

7 Conclusion

In this work, we present a novel activity-centric access control (ACAC) approach for smart and connected systems. Considering activity as the prime notion and abstraction to control, we propose an active and object-agnostic access control model, which captures the real-time and holistic context of the system to make an activity request decision. Focusing on the dependencies (D) among activities as one of the critical parameters, we formally develop a family of ACAC$_{\mathrm{D}}$ models supporting activity mutability. We also investigate the chain of dependencies (where dependent activities also can have dependencies) while changing the state of a mutable activity. Resolving chain of dependencies to accommodate the mutability of an activity may be challenging in terms of multiple dependency paths, race conditions and deadlock situations. We explain these challenges and propose potential solutions to deal with those. We also present a prototype implementation of ACAC$_{\mathrm{D}}$ sub-models with a comprehensive smart farming use case reflecting the use of combinations of ACAC$_{\mathrm{D}}$ sub-models and chain of dependencies. Performance is evaluated by the execution time to process many requests with different numbers of pre-, ongoing, and post-dependent activities' checks and updates.

In the future, we aim to extend this work to a fully mature ACAC model integrating all four authorizations (A), obligations (B), conditions (C), and dependencies (D) parameters. Moreover, our future direction includes developing a formal policy specification language incorporating the chain of dependencies along with other components and analyzing the reachability of incompatible activities as well. Further, a detailed performance evaluation in a real environment having different decision parameters will re-enforce the applicability of the ACAC model in large-scale smart systems.

Acknowledgements This research is partially supported by the NSF grants 2230609 and CREST-PRF 2112590.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

Compliance with Ethical Standards All authors confirm that the principles of ethical and professional conduct have been followed and declare that they have no Conflict of interest. In addition to that, the research work articulated in this manuscript does not contain any studies with human participants or animals performed by any of the authors.

Research Data Policy and Data Availability Statements The dataset generated and/or analyzed during the current study is available from the corresponding author on reasonable request.

References

1. Ameer, S., Benson, J., Sandhu, R.: The EGRBAC Model for Smart Home IoT. In: IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI), 457–462, (2020)
2. Schuster, R., Shmatikov, V., Tromer, E.: Situational access control in the internet of things. In: ACM SIGSAC Conference on Computer and Communications Security, pages 1056–1073, (2018)
3. Gusmeroli, S., Piccione, S., Rotondi, D.: A capability-based security approach to manage access control in the internet of things. Math. Comput. Model. **58**(5–6), 1189–1205 (2013)
4. Gupta, D., et al.: Access control model for Google cloud IoT. In: IEEE Conference on Big Data Security on Cloud, 198–208, (2020)
5. Gupta, M., Benson, J., Patwa, F., Sandhu, R.: Secure V2V and V2I communication in intelligent transportation using cloudlets. IEEE Trans. Serv. Comput. (2020)
6. Ameer, S., Sandhu, R.: The HABAC Model for Smart Home IoT and Comparison to EGRBAC. In: ACM Workshop on Secure and Trustworthy Cyber-Physical Systems, 39–48, (2021)
7. Lee, A.T., et al.: PARBAC: priority-attribute-based RBAC model for azure IoT cloud. IEEE Internet Things J. **7**(4), 2890–2900, (2020)
8. Bhatt, S., Pham, T.K., Gupta, M., Benson, J., Park, J., Sandhu, R.: Attribute-based access control for AWS internet of things and secure industries of the future. IEEE Access **9**, 107200–107223 (2021)
9. Gupta, M., Benson, J., Patwa, F., Sandhu, R.: Dynamic groups and attribute-based access control for next-generation smart cars. In: Proc. of the ACM Conference on Data and Application Security and Privacy, 61–72, (2019)
10. Xu, R., Chen, Y., Blasch, E., Chen, G.: A federated capability-based access control mechanism for internet of things (iots). In: Sensors and Systems for Space Applications XI, volume 10641, page 106410U. Int. Soc. Opt. Photonics (2018)
11. Park, J., Sandhu, R., Gupta, M., Bhatt, S.: Activity control design principles: next generation access control for smart and collaborative systems. IEEE Access **9**, 151004–151022 (2021)
12. Cathey, G., Benson, J., Gupta, M., Sandhu, R.: Edge Centric Secure Data Sharing with Digital Twins in Smart Ecosystems. In: IEEE TPS-ISA, (2021)
13. Colombo, P., Ferrari, E., Tümer, E.D.: Regulating data sharing across MQTT environments. JNCA **174**, 102907 (2021)
14. Gupta, M., Sandhu, R.: Towards activity-centric access control for smart collaborative ecosystems. In: Proceedings of the 26th ACM Symposium on Access Control Models and Technologies, 155–164, (2021)
15. Mawla, T., Gupta, M., Sandhu, R.: BlueSky: Activity Control: A Vision for "Active" Security Models for Smart Collaborative Systems. In: Proceedings of the 27th ACM on symposium on access control models and technologies, 207–216, (2022)
16. Nicklas, J.-P., Mamrot, M., Winzer, P., Lichte, D., Marchlewitz, S., Wolf, K.-D.: Use case based approach for an integrated consideration of safety and security aspects for smart home applications. In: 2016 11th System of Systems Engineering Conference (SoSE), 1–6. IEEE, (2016)
17. Khoussi, S., Mattas, A.: A brief introduction to smart grid safety and security. In: Handbook of system safety and security, 225–252. Elsevier, (2017)
18. Lacinák, M., Ristvej, J.: Smart city, safety and security. Procedia Eng. **192**, 522–527 (2017)
19. Tokody, D., Albini, A., Ady, L., Rajnai, Z., Pongrácz, F.: Safety and security through the design of autonomous intelligent vehicle systems and intelligent infrastructure in the smart city. Interdisciplinary Description of Complex Systems: INDECS, 16(3-A):384–396, (2018)

20. Threat Modeling | OWASP Foundation — owasp.org. https://owasp.org/www-community/Threat_Modeling. [Accessed 02-11-2023]
21. Thomas, R.K., Sandhu, R.S.: Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented authorization management. In: Database security XI, 166–181. Springer, (1998)
22. Park, J., Sandhu, R.: The UCON$_{\text{ABC}}$ usage control model. ACM Trans. Inf. Syst. Secur. (TISSEC) **7**(1), 128–174 (2004)
23. Park, J., Sandhu, R., Cheng, Y.: ACON: Activity-centric access control for social computing. In: IEEE ARES, 242–247, (2011)
24. Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering DAC, MAC and RBAC. In: IFIP Annual Conference on Data and Applications Security and Privacy, 41–55. Springer, (2012)
25. Gupta, M., others: An Attribute-Based Access Control for Cloud Enabled Industrial Smart Vehicles. IEEE Trans. Ind. Inf. (2020)
26. Bhatt, S., Sandhu, R.: ABAC-CC: Attribute-based access control and communication control for internet of things. In: Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, 203–212, (2020)
27. Sandhu, R., Park, J.: Usage control: A vision for next generation access control. In: International Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, 17–31. Springer, (2003)
28. Gupta, M., Sandhu, R., Mawla, T., Benson, J.: Reachability analysis for attributes in ABAC with group hierarchy. IEEE Trans. Dependable Secure Comput. **20**(1), 841–858 (2022)
29. Cho, M.-H., Lee, C.-H.: A low-power real-time operating system for ARC (actual remote control) wearable device. IEEE Trans. Consum. Electron. **56**(3), 1602–1609 (2010)
30. Yao, X., Farha, F., Li, R., Psychoula, I., Chen, L., Ning, H.: Security and privacy issues of physical objects in the IoT: Challenges and opportunities. Digital Communications and Networks, (2021)
31. Babun, L., et al.: A survey on IoT platforms: Communication, security, and privacy perspectives. Comput. Netw. **192**, 108040 (2021)
32. Ameer, S., Benson, J., Sandhu, R.: An attribute-based approach toward a secured smart-home IoT access control and a comparison with a role-based approach. Information **13**(2), 60 (2022)
33. Chen, Y., Meng, L., Zhou, H., Xue, G.: A blockchain-based medical data sharing mechanism with attribute-based access control and privacy protection. Wirel. Commun. Mob. Comput. (2021)
34. Zhang, Y., Yutaka, M., Sasabe, M., Kasahara, S.: Attribute-based access control for smart cities: a smart-contract-driven framework. IEEE Internet Things J. **8**(8), 6372–6384 (2020)
35. Sikder, A.K., et al.: Who's Controlling My Device? Multi-User Multi-Device-Aware Access Control System for Shared Smart Home Environment. ACM Trans. Internet of Things (2022)
36. Clark, S., et al.: ReLOG: A Unified Framework for Relationship-Based Access Control over Graph Databases. In: IFIP Annual Conference on Data and Applications Security and Privacy, 303–315. Springer, (2022)
37. Chakraborty, S., Sandhu, R.: On feasibility of attribute-aware relationship-based access control policy mining. In: IFIP Annual Conference on Data and Applications Security and Privacy, 393–405. Springer, (2021)
38. Arora, C.: Higher-Order (Temporal) Relationship-Based Access Control. Master's thesis, Science, (2022)
39. Bayreuther, S., Jacob, F., Grotz, M., Kartmann, R., et al.: BlueSky: Combining Task Planning and Activity-Centric Access Control for Assistive Humanoid Robots. In: Proc. of the 27th ACM SACMAT, 185–194, (2022)
40. Tan, L., Shi, N., Keping, Yu., Aloqaily, M., Jararweh, Y.: A blockchain-empowered access control framework for smart devices in green internet of things. ACM Trans. Internet Technol. (TOIT) **21**(3), 1–20 (2021)
41. Han, D., et al.: A blockchain-based auditable access control system for private data in service-centric IoT environments. IEEE Trans. Ind. Inf. (2021)
42. Qin, X., et al.: LBAC: A lightweight blockchain-based access control scheme for the internet of things. Inf. Sci. **554**, 222–235 (2021)
43. Algarni, S., et al.: Blockchain-based secured access control in an IoT system. Appl. Sci. **11**(4), 1772 (2021)
44. Nobi, M.N., et al.: Toward Deep Learning Based Access Control. In: Proc. of the ACM CODASPY, 143–154, (2022)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.