Access Control Synthesis for Physical Spaces

Petar Tsankov Mohammad Torabi Dashti David Basin
Department of Computer Science
ETH Zurich, Switzerland
{ptsankov, torabidm, basin}@inf.ethz.ch

***Abstract*—Access-control requirements for physical spaces, like office buildings and airports, are best formulated from a global viewpoint in terms of system-wide requirements. For example, "there is an authorized path to exit the building from every room." In contrast, individual access-control components, such as doors and turnstiles, can only enforce local policies, specifying when the component may open. In practice, the gap between the system-wide, global requirements and the many local policies is bridged manually, which is tedious, error-prone, and scales poorly.**

We propose a framework to automatically synthesize local access-control policies from a set of global requirements for physical spaces. Our framework consists of an expressive language to specify both global requirements and physical spaces, and an algorithm for synthesizing local, attribute-based policies from the global specification. We empirically demonstrate the framework's effectiveness on three substantial case studies. The studies demonstrate that access-control synthesis is practical even for complex physical spaces, such as airports, with many interrelated security requirements.

I. Introduction

Physical access control is used to restrict access to physical spaces. For example, it controls who can access which parts of an office building or how personnel can move within critical spaces such as airports or military facilities. As physical spaces are usually comprised of subspaces, such as rooms connected by doors, policies are enforced by multiple policy enforcement points (PEPs). Each PEP is associated to a control point, like a door, and enforces a *local* policy.

Consider, for example, an office building. An electronic door lock might control access to an office by enforcing a policy that states that only an *employee* may enter the office. This policy is local in the sense that its scope is limited to an individual enforcement point, here the office's door. The policy therefore does not guarantee that non-employees cannot enter the office, since the office may have other doors. Neither does it guarantee that employees can actually access the office. If employees cannot enter the corridor leading to the office's door, then the local policy is useless.

In contrast to the local policies for enforcement points, access-control requirements for physical spaces are typically *global*. They express constraints on the access paths through the entire space. In the example above, a requirement might be that employees should be able to access the office from the lobby. This requirement is global in that no single PEP alone can guarantee its satisfaction. A standard electronic lock, which enforces only local policies such as *grant access to employees*, is oblivious to the physical constraints of the office building and what policies the other PEPs enforce. It therefore cannot address this requirement.

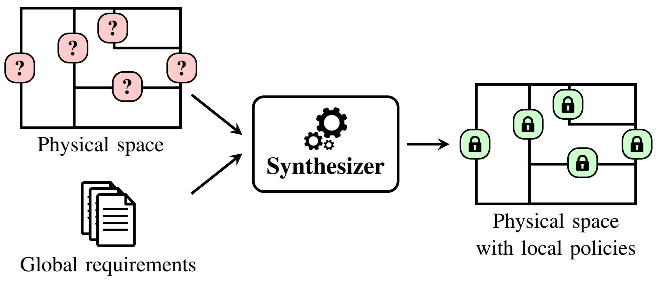


Fig. 1: Access control synthesis for physical spaces

Problem Statement. The discrepancy between global requirements and local policies creates an abstraction gap that must be bridged when configuring access-control mechanisms. We consider the problem of automatically synthesizing a set of PEP policies that together enforce global access-control requirements in a given physical space.

This problem is nontrivial. A given physical space usually constrains the ways subjects may access its subspaces. These constraints must be accounted for when configuring the individual PEPs. Moreover, global access-control requirements may have interdependencies and hence their individual solutions may not contribute to an overall solution. To illustrate this lack of compositionality, suppose in addition to the requirement that employees can access an office room from the lobby, we require that they must not enter the area where auditing documents are stored. Giving employees access to their office through *any* path satisfies the first requirement, but it would violate the second one if the path goes through the audit area.

In practice, constructing local policies for a physical space is a manual task where a security engineer writes individual policies, one per PEP, that collectively enforce the space's global requirements. This manual process results in errors, such as granting access to unauthorized subjects or denying access to authorized ones; the literature contains numerous examples of such problems [1]–[3]. Moreover, engineers must manually revise their policies whenever requirements are changed, or when the physical space changes, e.g. due to construction work. In short, writing local policies manually is error-prone and scales poorly. Our thesis is that it is also unnecessary: the automatic synthesis of local policies with system-wide security guarantees is a viable alternative.

Approach and Contributions. We propose a formal framework for automatically synthesizing local policies that run on distributed PEPs from a set of global access-control requirements for a given physical space. The framework's main ingredients are depicted in Figure 1. The key component is a *synthesizer*, which takes as input a model of the physical space and a set of global requirements. The synthesizer's output is the set of local policies that the PEPs enforce. If the global requirements are satisfiable, then the synthesizer is guaranteed to output a correct set of local policies; otherwise, it returns unsat to indicate that the requirements cannot be satisfied. Hence, using our framework, engineers can generate local policies from global requirements simply by formalizing the global requirements and modeling the physical space.

Below, we briefly describe the framework's components, depicted in Figure 1. We use directed graphs to model physical spaces: a node represents an enclosed space, such as an office or a corridor, and an edge represents a PEP, for example installed on a door or turnstile. The nodes are labeled to denote their attributes. These attributes may include the assets the node contains (audit documents), its physical attributes (international terminal), and its clearance level (high security zone). These attributes may be used when specifying policies. Formally, our model of a physical space is a Kripke structure.

We give a declarative language, called SpCtl, for specifying global requirements. Our language is built on the computation tree logic (CTL) [4] and supports subject attributes (e.g., an organizational role), time constraints (e.g., business-hour requirements), as well as quantification over paths and branches in physical spaces. To demonstrate its expressiveness, we show how common physical access-control requirements can be directly written in SpCtl. Moreover, to simplify the task of formalizing such requirements, we develop requirement patterns and illustrate their use through examples.

Our synthesis algorithm outputs attribute-based policies, expressed as constraints over subject attributes and contextual conditions, such as organizational roles and the current time. This covers a wide range of practical setups and scenarios, including attribute-based and role-based access control. We strike a balance between the requirement language's expressiveness and the complexity of synthesizing local policies. The synthesis problem we consider is NP-hard. However, we show that for practically-relevant requirements, it can be efficiently solved using existing SMT solvers. This is intuitively because physical spaces, in practice, induce directed graphs that have short simple-paths. We illustrate our framework's effectiveness using three case studies where we synthesize access-control policies for a university building, a corporate building, and an airport terminal. Synthesizing local policies in each case takes less than 30 seconds. The last two case studies are based on real-world examples developed together with KABA, a leading physical access control company.

To the best of our knowledge, ours is the first framework for synthesizing policies from system-wide access-control requirements. We thereby solve a fundamental problem in access control for physical spaces. An immediate practical consequence is that security engineers can focus on system-wide requirements, and delegate to our synthesizer the task of constructing the local policies with correctness guarantees. We remark that although this work is focused on access control for physical spaces, the ideas presented are general and can be extended to other domains, such as computer networks partitioned into subnetworks by distributed firewalls.

Organization. We give an overview of our access-control synthesis framework in Section II. In Section III, we describe and formalize our system model. In Section IV, we define our SpCtl language for specifying global requirements, and present requirement patterns. In Section V, we define the policy synthesis problem and prove its decidability. In Section VI, we define an efficient policy synthesis algorithm. In Section VII, we describe our implementation and report on our experiments. We review related work in Section VIII, and we draw conclusions and discuss future work in Section IX. Proofs can be found in the extended version of this paper [5].

II. Overview

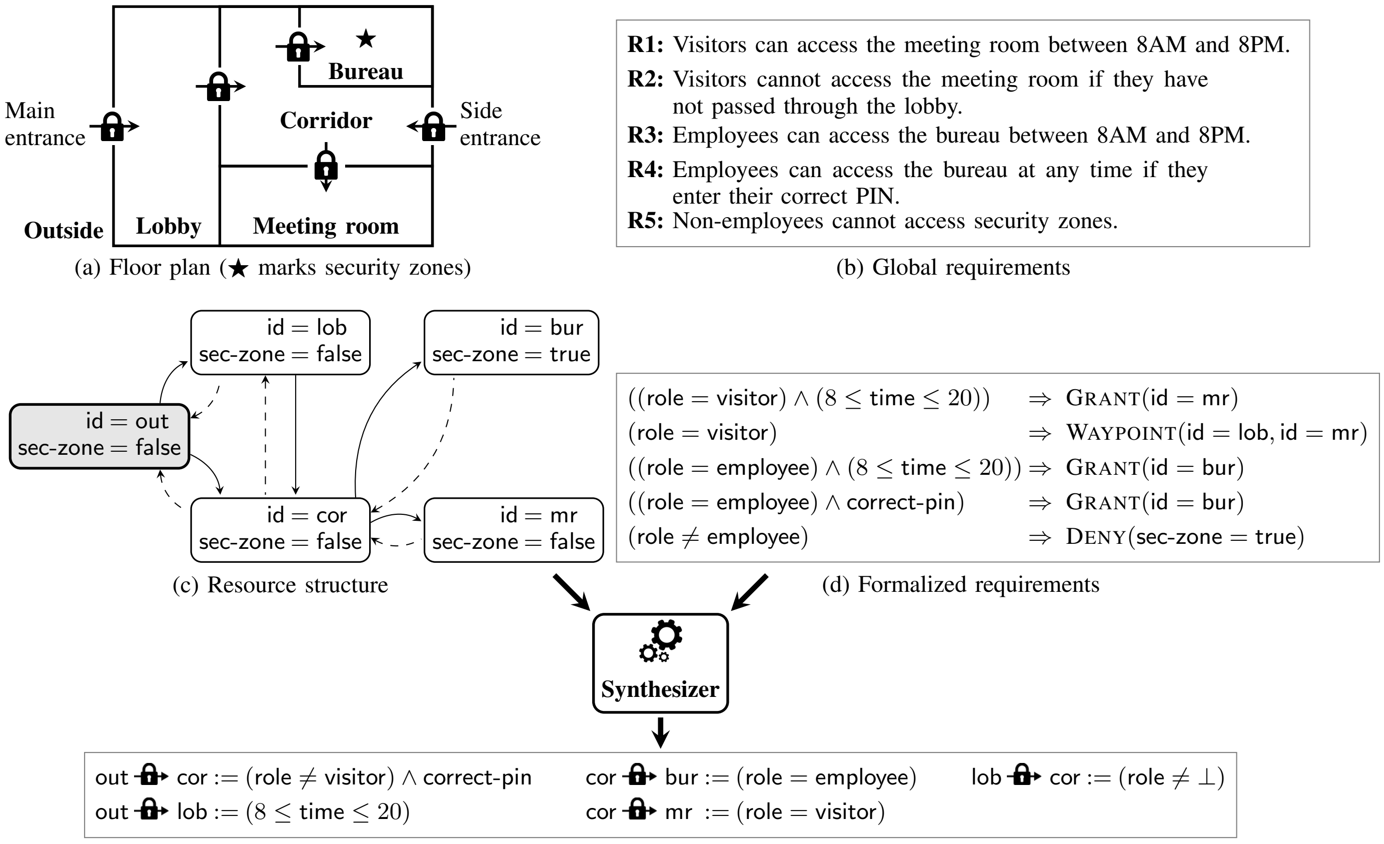
We start with a simple example that illustrates the challenges of constructing local policies that cumulatively enforce global access-control requirements. We also explain how our framework is used, that is, we describe its inputs and outputs.

A. Running Example

Consider a small office space consisting of a lobby, a bureau, a meeting room, and a corridor. The office layout is given in Figure 2(a). Access within this physical space is secured using electronic locks. Each door has a lock and a card reader. The lock stores a policy that defines who can open the door from the card reader's side. The door can be opened by anyone from the opposite side. We annotate locks with arrows in Figure 2(a) to indicate the direction that the locks restrict access. For example, the lock at the main entrance restricts who can access the lobby, and it allows anyone to exit the office space from the lobby. To open a door from the card reader's side, a subject presents a smartcard that stores the holder's credentials. The lock can access additional information, such as the current time, needed to evaluate the policy. The lock opens whenever the policy evaluates to grant.

The global requirements for this physical space are given in Figure 2(b). The requirements **R1**, **R3**, and **R4** define permissions, while **R2** and **R5** define prohibitions. To meet these requirements, the electronic locks must be configured with appropriate local policies. As previously observed, this is challenging because one must account for both spatial constraints and all global access-control requirements. We illustrate these points below.

Spatial Constraints. The layout of the physical space prevents subjects from freely requesting access to any resource. For example, the requirement **R1** is not met just because the meeting room's lock grants access to visitors; the visitor must also be able to enter the corridor from the outside. Such constraints must be accounted for when defining the local policies. To satisfy **R1**, we may for instance choose a path



(a) Floor plan (★ marks security zones)

R1: Visitors can access the meeting room between 8AM and 8PM.
R2: Visitors cannot access the meeting room if they have not passed through the lobby.
R3: Employees can access the bureau between 8AM and 8PM.
R4: Employees can access the bureau at any time if they enter their correct PIN.
R5: Non-employees cannot access security zones.

(b) Global requirements

(c) Resource structure

$((\mathsf{role} = \mathsf{visitor}) \wedge (8 \leq \mathsf{time} \leq 20))$	$\Rightarrow$	$\textsc{Grant}(\mathsf{id} = \mathsf{mr})$
$(\mathsf{role} = \mathsf{visitor})$	$\Rightarrow$	$\textsc{Waypoint}(\mathsf{id} = \mathsf{lob}, \mathsf{id} = \mathsf{mr})$
$((\mathsf{role} = \mathsf{employee}) \wedge (8 \leq \mathsf{time} \leq 20))$	$\Rightarrow$	$\textsc{Grant}(\mathsf{id} = \mathsf{bur})$
$((\mathsf{role} = \mathsf{employee}) \wedge \mathsf{correct\text{-}pin})$	$\Rightarrow$	$\textsc{Grant}(\mathsf{id} = \mathsf{bur})$
$(\mathsf{role} \neq \mathsf{employee})$	$\Rightarrow$	$\textsc{Deny}(\mathsf{sec\text{-}zone} = \mathsf{true})$

(d) Formalized requirements

out 🔒 cor := $(\mathsf{role} \neq \mathsf{visitor}) \wedge \mathsf{correct\text{-}pin}$
cor 🔒 bur := $(\mathsf{role} = \mathsf{employee})$
lob 🔒 cor := $(\mathsf{role} \neq \bot)$
out 🔒 lob := $(8 \leq \mathsf{time} \leq 20)$
cor 🔒 mr := $(\mathsf{role} = \mathsf{visitor})$

(e) Configuration assigning PEPs to local policies

Fig. 2: Synthesizing the local policies for our running example

from the main entrance to the meeting room and configure all the locks along that path to grant access to visitors.

Global Requirements. Each global requirement typically has multiple sets of local policies that satisfy it. The local policies must however be constructed to ensure that *all* requirements are satisfied *simultaneously*. For example, the requirement **R1** is satisfied if the side-entrance lock and the meeting room lock both grant access to visitors between 8AM and 8PM. It can also be satisfied by ensuring that the main entrance, the lobby, and the meeting room locks all grant access to visitors between 8AM and 8PM. Granting visitors access through the side entrance however violates the requirement **R2**, which requires that visitors pass through the lobby. Hence, to meet both requirements, the locks along the path through the lobby must grant access to visitors between 8AM and 8PM, while the side-entrance lock must always deny access to visitors.

B. Synthesis Framework

Figure 2(c-e) depicts our framework's input and output for our running example. The input is a model of the physical space and a specification of its global requirements. The output produced by our synthesizer is a set of local policies.

A physical space is modeled as a rooted directed graph called a *resource structure*. We have depicted the root node in gray. In our example, this corresponds to the public space that surrounds the office space, e.g. public streets. The remaining (non-root) nodes are the spaces inside the building. The locks control access along the edges. A subject can traverse a solid edge of the resource structure only if the lock's policy evaluates to grant, whereas any subject can follow the dashed edges. Hence, the locks effectively enforce the grant-all policy along the dashed edges. We use two attributes to label the physical spaces: the attribute id represents room identifiers, and sec-zone formalizes that a space is inside the security zone.

Global requirements are specified using a declarative language, called SPCTL. In Figure 2(d) we show the formalization of our running example's requirements in SPCTL. For instance, **R1**, which states that visitors can access the meeting room between 8AM and 8PM, is formalized as $\big((\mathsf{role} = \mathsf{visitor}) \wedge (8 \leq \mathsf{time} \leq 20)\big) \Rightarrow \textsc{Grant}(\mathsf{id} = \mathsf{mr})$. This formalization instantiates SPCTL's permission pattern GRANT to state that there is a path from outside to the meeting room such that every lock on the path grants access to any visitor between 8AM and 8PM. We define SPCTL's syntax and semantics and present several patterns in Section IV.

Given these inputs, the synthesizer automatically constructs a local policy for each lock. The synthesized policies are attribute-based policies that collectively enforce the global requirements. The synthesized policies for our running example are given in Figure 2(e). We write, for example, cor 🔒 bur for the synthesized policy deployed at the bureau's lock. This policy (role = employee) grants access to subjects with the role employee. We define the synthesis problem, and the syntax and semantics of attribute-based local policies in Section V.

III. Physical Access Control

A. Basic notions

In this section, we formalize our system model for physical access control. Our terminology is based, in part, on the XACML reference architecture [6].

Each physical space is partitioned into finitely many enclosed spaces and one open (public) space. We call the enclosed spaces *resources*. Two spaces may be directly connected with a *gate*, controlled by a *policy enforcement point* (PEP). Examples of gates include doors, turnstiles, and security checkpoints. Each PEP has its own *policy decision point* (PDP), which stores a *local policy* mapping access requests to access decisions. Each *access request* consists of subject credentials, which the PDP receives from the PEP, as well as contextual attributes, if needed, obtained from *policy information points* (PIPs). The PIPs are distributed information sources that provide contextual attributes required by the PDP for access decisions. Examples of PIPs include revocation list servers and secure time servers.

To enter a space, a subject provides his credentials to the PEP that controls the gate. The PEP forwards the subject's credentials to its PDP. The PDP, in turn, queries the PIP if needed, evaluates the policy, and then forwards the access decision — either grant or deny — to the PEP. The PEP then enforces the PDP's decision. See Figure 3.

We assume that access requests contain all relevant information for making access decisions. PDPs can thus make their decisions independent of past access requests. Hence it has no bearing on our model whether the PDPs are actually distributed or are realized through a centralized system. This is desirable from a practical standpoint since PDPs and PIPs need not be equipped with logging mechanisms. Moreover, different PDPs and PIPs need not synchronize their local views on the request history. In this sense they are autonomous entities.

The access requests and local policies we consider are attribute based and may reference three kinds of attributes. A *subject attribute* contains information about a subject. For example, Alice's *organizational role* and *clearance level* are her subject attributes. Subjects can provide PEPs with their attributes in the form of credentials. A *contextual attribute* represents information about the security context provided by a PIP, such as the list of revoked credentials and the current time. We also introduce *resource attributes*, which represent information about resources. For example, the attributes *floor* and *department* may represent the floor of an office space and the department it belongs to. We use resource attributes to specify global requirements. They are however not needed for expressing access requests or local policies in our model. This is because the PDPs associated to any resource can be hardwired with all the attributes of that resource. In this sense, each PDP "knows" the space under its control.

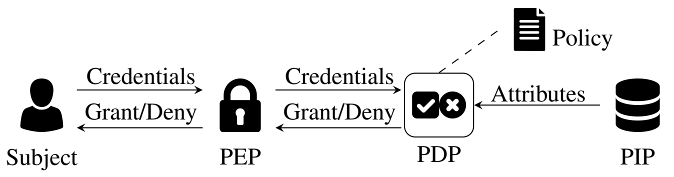


Fig. 3: System model

Our system model targets electronic PEP/PDPs that can enforce attribute-based policies and read digital credentials, e.g. stored on smart cards and mobile phones. Manufacturers often refer to these as smart locks [7]–[9]. In large physical access-control systems, smart locks are rapidly replacing mechanical locks and keys, which can only enforce simple, crude policies.

B. Formalization

We now formalize the above notions.

Attributes. Fix a finite set $\mathcal{A}$ of *attributes* and a set $\mathcal{V}$ of attribute *values*. The *domain* function $\mathsf{dom}\colon \mathcal{A} \to \mathcal{P}(\mathcal{V})$ associates each attribute with the set of values it admits. For instance, the current time attribute is associated with the set of natural numbers, and the clearance level attribute is associated with a fixed finite set of levels. We assume that any attribute can take the designated value $\perp$, representing the situation where the attribute's value is unknown. We partition the set of attributes into subject attributes $\mathcal{A}_S$, contextual attributes $\mathcal{A}_C$, and resource attributes $\mathcal{A}_R$.

Access Requests. We represent an access request as a total function that maps subject and contextual attributes to values from their respective domains. This function is computed by PDPs after receiving a subject's credentials and querying PIPs. For instance, the PDP maps the attribute role to visitor when the subject's credentials indicate this. It maps the attribute correct-pin to true when the PIN entered through the keypad attached to the PDP is correct. Finally, it maps the contextual attribute time to 8 after querying a time server at 8AM. We denote the set of all access requests by $\mathcal{Q}$.

A remark on set-valued attributes is due here. In some settings, attributes take a finite set of values, as opposed to a single value. For example, in role-based access control, a subject may activate multiple roles. The attribute role must then be assigned with the set of all the activated roles. We account for such set-valued attributes simply by defining a Boolean attribute for each value; for example, we define role_employee and role_manager. An access request q assigns true to both Boolean attributes whenever a subject has activated both the employee and the manager roles.

Local Policies. Local policies map access requests to grant or deny. We extensionally define local policies as subsets of $\mathcal{Q}$: a local policy is defined as the set of requests that it grants. The structure $(\mathcal{P}(\mathcal{Q}), \subseteq, \cap, \cup, \emptyset, \mathcal{Q})$ is a complete lattice that orders local policies by their permissiveness. The least permissive policy, namely $\emptyset$, denies all access requests, and the most permissive one, i.e. $\mathcal{Q}$, grants them all. In section V-A, we will intensionally define local policies as constraints over subject

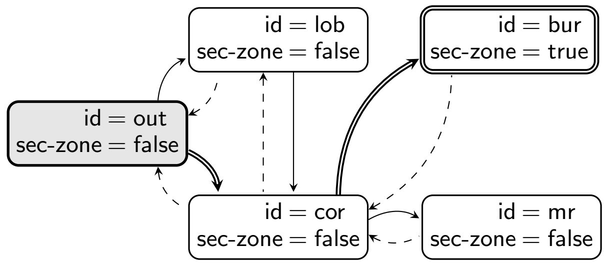


Fig. 4: The double-lined edges denote the PEPs that deny the access request $q = \{\mathsf{role} \mapsto \mathsf{visitor}, \mathsf{time} \mapsto 10, \mathsf{correct\text{-}pin} \mapsto \bot\}$, given the configuration c from Figure 2. The resource structure $S_{c,q}$ is obtained by removing the double-lined edges and nodes.

and contextual attributes. The local policies shown in Figure 2, for example, are defined by such constraints.

Resource Structures. We now give a formal model of physical spaces. A *resource structure* is a tuple $S = (\mathcal{R}, E, r_e, L)$, where $\mathcal{R}$ is a set of *resources*, $E \subseteq \mathcal{R} \times \mathcal{R}$ is an irreflexive edge relation, $r_e \in \mathcal{R}$ is the *entry resource*, and $L : \mathcal{R} \to (\mathcal{A}_R \to \mathcal{V})$ is a total function mapping resources to resource attribute valuations. We assume that every resource $r \in \mathcal{R}$ is reachable from the entry resource r_e, that is, $(r_e, r) \in E^*$, where E^* is the reflexive-transitive closure of E.

The edges in a resource structure model PEPs. The irreflexivity of E captures the condition that once a subject enters a physical space, he cannot re-enter the space before first leaving it. We assume that resource structures do not contain *deadlocks*. A resource r_0 in S is a deadlock if there does not exists an r_1 such that $(r_0, r_1) \in E$. This assumption is valid in physical-space access control: a deadlock resource corresponds to a "black hole" that no one can leave. Note that dead-end corridors are not deadlocks, provided one can backtrack.

The entry resource r_e represents the public space and the remaining resources denote enclosed spaces. A resource structure describes how subjects can access resources. A subject accesses a resource along a path, which is a sequence of resources connected by edges, starting from the entry resource. For example, before entering a room in a hotel, a subject enters the hotel's lobby from the street, and then goes through the corridor. Figure 2(c) gives an example of a resource structure.

Configurations. Each edge of a resource structure represents a gate controlled by a local policy installed on the gate's PDP. We therefore define a *configuration* for a resource structure S as a function that assigns to each edge of S a local policy. We write C_S for the set of all configurations for S. The set C_S is partially ordered under the relation $\sqsubseteq_S$, defined as: $c \sqsubseteq_S c'$ if for any edge e of S we have $c(e) \subseteq c'(e)$. Namely, a configuration is less permissive than another configuration if, for any edge, the former assigns a less permissive local policy than the latter.

We can now define which resources are accessible given an access request and a configuration. For a resource structure S, a configuration c for S, and an access request q, we define $S_{c,q}$ as the resource structure obtained by removing all the edges from S whose policies deny q, and then removing all nodes that are not reachable from the entry resource. The structure $S_{c,q}$'s entry resource is the same as S's. To illustrate, consider the resource structure S and the configuration c given in Figure 2, and the access request $q = \{\mathsf{role} \mapsto \mathsf{visitor}, \mathsf{time} \mapsto 10, \mathsf{correct\text{-}pin} \mapsto \bot\}$. The side-entrance PEP and the bureau PEP deny q and therefore these two edges are removed from S. The node that represents the bureau is not reachable from the entry resource and it is thus also removed. In Figure 4 we depict the removed edges and nodes.

We remark that the structure $S_{c,q}$ is defined for a fixed access request q. Access requests, which assign values to subject and contextual attributes, can however change, for instance when a subject's role is revoked or as time progresses. We abstract away such changes in $S_{c,q}$'s definition. In our running example, this amounts to assuming that a subject's role does not change during this time, and the time needed to move through the office building is negligible compared to the time needed for a subject's access rights to change; for example, the requirements **R1-5** stipulate that subject's access rights may change only twice per day — at 8AM and at 8PM. This abstraction corresponds to taking a snapshot of all the attributes, and then computing $S_{c,q}$ based on the snapshot. We refer to these snapshots as *sessions*. Henceforth we interpret global requirements and local policies in the context of such sessions.

Interpreting requirements and polices in the context of a session is justified for many practical scenarios. This is because, in most practical settings, changes in subject and contextual attributes are addressed through out-of-band mechanisms. To illustrate, consider a subject who has the role visitor and enters the meeting room of our running example at 3PM as permitted by the system's requirements. Now, suppose that the subject's visitor role is revoked at 4PM, or that the subject remains in the meeting room until 10PM. No access-control system can force the subject to leave. In practice, out-of-band mechanisms, such as security guards, address such concerns.

In the following sections, we confine our attention to configurations that do not introduce deadlocks. That is, we consider those configurations c where for any $q \in \mathcal{Q}$, the structure $S_{c,q}$ is deadlock-free. In Section IV-C, we describe how this provision can be encoded as a global requirement.

IV. Specifying Requirements

In this section we define SpCtl, a simple declarative language for specifying requirements. We give the language's syntax and semantics in Section IV-A. To simplify the specification of global requirements, in Section IV-B we present four requirement patterns that capture common access-control idioms for physical spaces. Finally, in Section IV-C, we illustrate the specification of two generic access-control requirements: deny-by-default and deadlock-freeness.

A. Requirement Specification Language

The design of SpCtl has been guided by real-world physical access-control requirements. Virtually all such require-

$$\begin{array}{rcl} a = c & := & a \in \{c\} \\ a \neq c & := & \neg(a = c) \\ a_{\mathsf{bool}} & := & a_{\mathsf{bool}} = \mathsf{true} \\ a_{\mathsf{num}} \leq n & := & a_{\mathsf{num}} \in \{0, \ldots, n\} \\ a_{\mathsf{num}} \geq n & := & \neg(a_{\mathsf{num}} \leq n-1) \\ n \leq a_{\mathsf{num}} \leq n' & := & (a_{\mathsf{num}} \geq n) \wedge (a_{\mathsf{num}} \leq n') \end{array}$$

Fig. 5: Syntactic shorthands: $a \in \mathcal{A}$ is an attribute, $a_{\mathsf{num}} \in \mathcal{A}_{\mathsf{num}}$ is a numeric attribute, $a_{\mathsf{bool}} \in \mathcal{A}_{\mathsf{bool}}$ is a boolean attribute, $n, n' \in \mathbb{N}$ are natural numbers.

ments can be formalized as properties that specify which physical spaces subjects can and cannot access, directly and over paths, based on the security context and on the physical spaces they have accessed. In our physical access-control model, subjects choose which physical spaces to access, which induces a branching structure over the spaces they access. We therefore build our requirement specification language SpCtl upon the computation tree logic (CTL) [10], whose branching semantics is a natural fit for physical spaces.

Syntax. A requirement specified in SpCtl is a formula of the form $T \Rightarrow \varphi$ given by the following BNF:

$$\begin{array}{rcl} T & ::= & \mathsf{true} \mid a_s \in D \mid a_c \in D \mid \neg T \mid T \wedge T \\ \varphi & ::= & \mathsf{true} \mid a_r \in D \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{EX}\varphi \mid \mathsf{AX}\varphi \\ & \mid & \mathsf{E}[\varphi \mathsf{U} \varphi] \mid \mathsf{A}[\varphi \mathsf{U} \varphi] \ . \end{array}$$

Here $a_s \in \mathcal{A}_S$ is a subject attribute, $a_c \in \mathcal{A}_C$ is a contextual attribute, $a_r \in \mathcal{A}_R$ is a resource attribute, and $D \subseteq \mathcal{V}$ is a finite subset of values. The formula T is a constraint over subject and contextual attributes that defines the access requests to which the requirement applies. We call T the *target*. The formula φ is a CTL formula over resource attributes. It defines a path property that must hold for all access requests to which the requirement is applicable. We call φ an *access constraint*.

Note that additional Boolean and CTL operators can be defined in the standard way. For example, we write false for $\neg$true, and define the Boolean connectives $\vee$ and $\Rightarrow$ in the standard manner using $\neg$ and $\wedge$. We will later make use of the CTL operators $\mathsf{EF}\varphi$, $\mathsf{AG}\varphi$, and $\mathsf{A}[\varphi \mathsf{R} \psi]$, which are defined as $\mathsf{E}[\mathsf{true}\ \mathsf{U}\ \varphi]$, $\neg(\mathsf{EF}\neg\varphi)$, and $\neg(\mathsf{E}[\neg\varphi \mathsf{U} \neg\psi])$, respectively. Below we give intuitive explanations of EX, AX, EU, and AU, which are standard CTL connectives.

The connectives *exists-next* EX and *always-next* AX constrain the physical spaces that a subject can access next. In our running example, suppose that a subject has entered the lobby. The subject can next enter the corridor or go to the public space: these are immediately accessible from the lobby. In the lobby, $\mathsf{EX}\varphi$ states that the formula φ is true in at least one of these "next" spaces. In contrast, $\mathsf{AX}\varphi$ states that φ is true both in the corridor and in the public space.

The operators *exists-until* EU and *always-until* AU relate two access constraints φ_1 and φ_2 over paths. The formula $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ states that there exists a path that reaches a resource r that satisfies φ_2, and any resource prior to r on the path satisfies φ_1. We use this connective to formalize, for example, waypointing requirements such as: visitors cannot access the meeting room until they have accessed the lobby. The formula $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ states that every path reaches some resource r that satisfies φ_2, and that any resource prior to r on the path satisfies φ_1.

$$\begin{array}{lll} S, r_0 \models \mathsf{true} & & \\ S, r_0 \models a \in D & \text{if} & L(r_0)(a) \in D \\ S, r_0 \models \neg\varphi & \text{if} & S, r_0 \not\models \varphi \\ S, r_0 \models \varphi_1 \wedge \varphi_2 & \text{if} & S, r_0 \models \varphi_1 \text{ and } S, r_0 \models \varphi_2 \\ S, r_0 \models \mathsf{EX}\varphi & \text{if} & \exists (r_0, r_1, \cdots) \in S(r_0).\ S, r_1 \models \varphi \\ S, r_0 \models \mathsf{AX}\varphi & \text{if} & \forall (r_0, r_1, \cdots) \in S(r_0).\ S, r_1 \models \varphi \\ S, r_0 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2] & \text{if} & \exists (r_0, r_1, \cdots) \in S(r_0).\ \exists i \geq 0. \\ & & S, r_i \models \varphi_2 \wedge \forall j \in [0, i).\ S, r_j \models \varphi_1 \\ S, r_0 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2] & \text{if} & \forall (r_0, r_1, \cdots) \in S(r_0).\ \exists i \geq 0. \\ & & S, r_i \models \varphi_2 \wedge \forall j \in [0, i).\ S, r_j \models \varphi_1 \end{array}$$

Fig. 6: The relation $\models$ between a resource structure $S = (\mathcal{R}, E, r_e, L)$, a resource $r_0 \in \mathcal{R}$, and an access constraints φ.

To simplify writing attribute constraints in SpCtl, we introduce in Figure 5 abbreviations for numeric and boolean attributes. Based on the attributes' domains, we partition the set of attributes $\mathcal{A}$ into *numeric attributes* A_{num}, *boolean attributes* A_{bool}, and *enumerated attributes* A_{enum}: An attribute a is *numeric* if $\mathsf{dom}(a) = \mathbb{N} \cup \{\bot\}$; it is *boolean* if $\mathsf{dom}(a) = \{\mathsf{false}, \mathsf{true}, \bot\}$; otherwise, it is enumerated and $\mathsf{dom}(a)$ is finite. We may write a_{num} or a_{bool} to emphasize that an attribute a is numeric or boolean, respectively.

Semantics. We first inductively define the satisfaction relation $\vdash$ between an access request $q \in \mathcal{Q}$ and a target:

$$\begin{array}{llll} q & \vdash & \mathsf{true} & \\ q & \vdash & a \in D & \text{if} \quad q(a) \in D \\ q & \vdash & \neg T & \text{if} \quad q \not\vdash T \\ q & \vdash & T_1 \wedge T_2 & \text{if} \quad q \vdash T_1 \text{ and } q \vdash T_2 \ . \end{array}$$

A requirement $T \Rightarrow \varphi$ is *applicable* to an access request q iff q satisfies the target T, i.e. $q \vdash T$. For example, the requirement $(\mathsf{role} = \mathsf{visitor}) \Rightarrow \varphi$ is applicable to all access requests that assign the value visitor to the subject attribute role.

Let $S = (\mathcal{R}, E, r_e, L)$ be a resource structure. A *path* of S is an infinite sequence of resources $(r_0, r_1, \cdots)$ such that $\forall i \geq 0.\ (r_i, r_{i+1}) \in E$, and we denote the set of all paths rooted at a resource r_0 by $S(r_0)$. In Figure 6, we inductively define the satisfaction relation $\models$ between a resource structure, a resource, and an access constraint. A resource structure S with an entry resource r_e *satisfies* an access constraint φ, denoted by $S \models \varphi$, iff $S, r_e \models \varphi$.

Definition 1. *Let S be a resource structure, c a configuration for S, and $T \Rightarrow \varphi$ a requirement. S configured with c* satisfies *$T \Rightarrow \varphi$, denoted by $S, c \Vdash (T \Rightarrow \varphi)$, iff $q \vdash T$ implies $S_{c,q} \models \varphi$, for any access request $q \in \mathcal{Q}$.*

We extend $\Vdash$ to sets of requirements as expected. Given a set of requirements $R = \{T_1 \Rightarrow \varphi_1, \ldots, T_n \Rightarrow \varphi_n\}$, a resource structure S configured with c satisfies R, denoted by $S, c \Vdash R$, iff $S, c \Vdash (T_i \Rightarrow \varphi_i)$ for all i, $1 \leq i \leq n$.

Pattern	Shorthand	Specification	Description	Intuitive Semantics
Permission	$T \Rightarrow \textsc{Grant}(\varphi)$	$T \Rightarrow \mathsf{EF}\ \varphi$	T-requests can access φ-spaces.	$\boxed{r_e} \xrightarrow{✓} \boxed{\varphi}$
Prohibition	$T \Rightarrow \textsc{Deny}(\varphi)$	$T \Rightarrow \mathsf{AG}(\neg\varphi)$	T-requests cannot access φ-spaces.	$\boxed{r_e} \xrightarrow{✗} \boxed{\varphi}$
Blocking	$T \Rightarrow \textsc{Block}(\varphi, \psi)$	$T \Rightarrow \mathsf{AG}(\varphi \Rightarrow \mathsf{AG}(\neg\psi))$	T-requests cannot access a ψ-space after accessing a φ-space.	$\boxed{r_e} \longrightarrow \boxed{\varphi} \xrightarrow{✗} \boxed{\psi}$
Waypointing	$T \Rightarrow \textsc{Waypoint}(\varphi, \psi)$	$T \Rightarrow \mathsf{A}[\varphi \mathsf{R} \psi]$	T-requests must access a φ-space before accessing a ψ-space.	$\begin{tikzcd} \boxed{r_e} \arrow[r] \arrow[rr, bend left, "✗"] & \boxed{\varphi} \arrow[r] & \boxed{\psi} \end{tikzcd}$

Fig. 7: SpCtl Patterns: The entry resource r_e in the intuitive semantics is depicted using a gray rectangle. The arrows $\varphi \xrightarrow{✓} \psi$ $\big(\varphi \xrightarrow{✗} \psi\big)$ indicate that there must (must not) exist a path from a φ-space to a ψ-space along which T-requests are granted.

We remark that resource structures can easily be represented using standard Kripke structures [10] by mapping each resource to a Kripke state and each resource attribute valuation to sets of atomic propositions. The access constraints can be similarly mapped to standard CTL formulas by translating attribute constraints into propositional logic. Note however that while Kripke structures are often used to represent changes of, say, a concurrent system's state over time, resource structures model static physical spaces.

B. Requirement Patterns

SpCtl can be directly used to specify global requirements. However, to illustrate its use and expressiveness, we present the formalization of common physical access-control idioms.

We have studied the requirements of an airport, a corporate building, and a university campus to elicit the common structure of physical access-control requirements. To distill the basic requirement patterns, we split complex requirements into their atomic parts. Our analysis revealed four common patterns, which we formalize below. The first pattern abstracts *positive* requirements, which stipulate that the access-control system must grant certain access requests. The remaining three patterns capture *negative* requirements, which stipulate that the access-control system must deny certain access requests.

We use the following terminology when describing requirements. Given a target T, we call an access request q a T-*request* if $q \vdash T$, i.e. q satisfies the target T. Given a resource structure S and an access constraint φ, we say that a subject can access a φ*-space* of S if the subject can access a physical space r_0 of S such that $S, r_0 \models \varphi$, i.e. the space r_0 satisfies the access constraint φ. Our patterns are summarized in Figure 7.

Permission. The *permission* pattern abstracts requirements stating that T-requests can access φ-spaces from the entry resource. Permission requirements have the form $T \Rightarrow (\mathsf{EF}\ \varphi)$. The exists-future operator EF formalizes that a φ-space is reachable from the entry resource. For example, the requirement **R3** stipulating that employees can access the bureau between 8AM and 8PM is formalized as

$$\big((\mathsf{role} = \mathsf{employee}) \wedge (8 \leq \mathsf{time} \leq 20)\big) \Rightarrow \mathsf{EF}\ (\mathsf{id} = \mathsf{bur}).$$

The target $(\mathsf{role} = \mathsf{employee}) \wedge (8 \leq \mathsf{time} \leq 20)$ formalizes that this requirement is applicable only to access requests made by visitors at times between 8AM and 8PM. The access constraint $\mathsf{EF}(\mathsf{id} = \mathsf{bur})$ is satisfied iff the resource structure has a path from the entry resource to the bureau. The requirements **R1** and **R4** of our running example are also instances of the permission pattern.

Prohibition. Dual to the permission pattern, the *prohibition* pattern captures requirements stating that T-access requests cannot access a φ-space. Prohibition requirements have the form $T \Rightarrow \mathsf{AG}(\neg\varphi)$. The operator AG quantifies over all paths reachable from the entry resource. An example taken from our airport requirements is: Passengers cannot access the departure gate zones without a boarding pass. Another example is requirement **R5**, formalized as

$$(\mathsf{role} \neq \mathsf{employee}) \Rightarrow \mathsf{AG}(\neg\, \mathsf{sec\text{-}zone})\ .$$

The target $\mathsf{role} \neq \mathsf{employee}$ is satisfied by access requests that assign a value other than $\mathsf{employee}$ to the attribute role. The access constraint $\mathsf{AG}(\neg\, \mathsf{sec\text{-}zone})$ is satisfied if no path leads to a security zone.

Blocking. The *blocking* pattern captures requirements stating that subjects cannot access a ψ-space after they have accessed a φ-space. Intuitively, accessing a φ-space *blocks* the subject from accessing ψ-spaces. At international airports, for example, passengers may not access departure gate zones after they have accessed the baggage claim. Blocking requirements have the form $T \Rightarrow \mathsf{AG}(\varphi \Rightarrow \mathsf{AG}(\neg\psi))$. The airport example is formalized as:

$$\begin{aligned}(\mathsf{role} = \mathsf{passenger}) \Rightarrow \mathsf{AG}\big(&(\mathsf{zone} = \mathsf{baggage\text{-}claim})\\ &\Rightarrow \mathsf{AG}\ \neg(\mathsf{zone} = \mathsf{departure})\big)\ .\end{aligned}$$

This requirement instantiates the blocking pattern: the target T is $(\mathsf{role} = \mathsf{passenger})$, and the two access constraints ψ and φ are $(\mathsf{zone} = \mathsf{departure})$ and $(\mathsf{zone} = \mathsf{baggage\text{-}claim})$.

Waypointing. The *waypointing* pattern captures requirements stipulating that subjects must first access a φ-space before accessing a ψ-space. For example, passengers cannot access an airport's terminal before they have passed through a security check. This is a negative requirement that restricts how passengers can access the terminal. Waypointing requirements have the form $T \Rightarrow (\mathsf{A}[\varphi \mathsf{R} \psi])$. The globally-release operator AR quantifies over all paths from the entry resource and formalizes that if ψ holds at some point, then φ was valid at least once beforehand. The requirement **R2** of our running example is an instance of the waypointing pattern and is formalized as

$$(\mathsf{role} = \mathsf{visitor}) \Rightarrow \mathsf{A}[(\mathsf{id} = \mathsf{lob})\mathsf{R}(\mathsf{id} = \mathsf{mr})] \ .$$

The target specifies that this requirement applies to all access requests made by visitors. The access constraint is satisfied if all paths to the meeting room go through the lobby.

The four idioms just described cover *all* the requirements that arose in the case studies that we report on in Section VII-B. We remark though that there are global requirements that are not instances of these four patterns. For example, in corporate buildings, a subject must be able to access the parking lot if he or she has access to an office. Although this requirement cannot be expressed using the above patterns, it can be directly formalized in SPCTL as follows:

$$\mathsf{true} \Rightarrow \big((\mathsf{EF}(\mathsf{zone} = \mathsf{office})) \Rightarrow (\mathsf{EF}(\mathsf{id} = \mathsf{parking\text{-}lot}))\big) \ .$$

In general, as SPCTL supports all CTL operators, it can specify any branching property expressible in CTL.

C. Generic Requirements

We now describe two commonly-used generic requirements.

Deny-by-default. The deny-by-default principle stipulates that if an access request can be denied without violating the requirements, then it should be denied; cf. [11]. Security engineers often follow this principle to avoid overly permissive local policies. To illustrate, consider our running example and imagine that the role intern is contained in the domain of the attribute role. The requirements given in Figure 2(b) do not prohibit an intern from accessing, say, the meeting room. However, denying interns access to the meeting room also complies with these requirements.

The following requirement, called *deny-by-default*, instantiates the above principle: If no positive requirement is applicable to an access request, then only the entry space is accessible to the subject who makes such a request. To formalize this requirement, we first define positive and negative requirements. Let c and c' be two configurations for a given resource structure S. A requirement $T \Rightarrow \varphi$ is *positive* if $S, c \Vdash (T \Rightarrow \varphi)$ and $c \sqsubseteq_S c'$ imply $S, c' \Vdash (T \Rightarrow \varphi)$. A requirement $T \Rightarrow \varphi$ is *negative* if $S, c \Vdash (T \Rightarrow \varphi)$ and $c' \sqsubseteq_S c$ imply $S, c' \Vdash (T \Rightarrow \varphi)$. Intuitively, if a configuration satisfies a positive (negative) requirement, then any more (less) permissive configuration also satisfies the requirement. We remark that although not all requirements are positive or negative, most real-world requirements are, including all requirements specified in this paper.

Let R be a set of requirements that contains only positive and negative requirements, and let $\{T_1 \Rightarrow \varphi, \cdots, T_n \Rightarrow \varphi_n\}$ be the set of all positive requirements contained in R. The deny-by-default requirement for R is

$$(\neg T_1) \wedge \cdots \wedge (\neg T_n) \Rightarrow \mathsf{AX}\ (\mathsf{id} = \mathsf{entry}) \ .$$

Here we assume that $L(r_e)(\mathsf{id}) = \mathsf{entry}$, i.e. the entry resource r_e is labeled with entry. Adding this requirement to our running example's requirements would ensure that an intern cannot access, for example, the meeting room. Note that if R contains no positive requirements, then the default-by-requirement is true $\Rightarrow$ AX (id = entry), which formalizes that all subjects can only access the entry space.

Deadlock-freeness. A *deadlock-freeness* requirement stipulates that there are no deadlocks in a system, i.e. resources that a subject can access and then never leave. For example, the meeting room of our running example would be a deadlock if visitors could enter it, but never leave. As discussed in our system model, local policies that introduce deadlocks are undesirable.

Formally, the deadlock-freeness requirement is defined as:

$$\mathsf{true} \Rightarrow \mathsf{AG}\ \mathsf{EX}\ \mathsf{true} \ .$$

This requirement applies to all access requests. The access constraint AG EX true states that for any resource a subject can access, there is a resource that the subject can access next. A resource structure S and a configuration c satisfy this requirement iff for any access requests $q \in \mathcal{Q}$, $S_{c,q}$ has no deadlocks.

V. Policy Synthesis Problem

We now define the policy synthesis problem. We show that this problem is decidable but NP-hard.

A. Problem

Definition 2. *The* policy synthesis problem *is as follows:*

Input. *A resource structure S and a set of requirements R.*

Output. *A configuration c such that $S, c \Vdash R$, if such a configuration exists, and* unsat *otherwise.*

The synthesized configuration defines the local policies to be deployed at the PEPs. Recall that a policy is extensionally defined as the set of access requests for which the PEP grants access. As such a set may, in general, be infinite, one cannot simply output an extensional definition of the synthesized configuration. We therefore define local policies intensionally by constraints over subject and contextual attributes, expressed in the same language that we specify requirement targets in Section IV. We remark that policies and targets are often formalized with the same language; cf. [12], [13]. The semantics of an intensional local policy P is then simply the function that maps an access request q to grant if $q \vdash P$ and to deny otherwise. Figure 2 illustrates the input and output to the policy synthesis problem for our running example.

An example of a local policy defined over the attributes role and time is $(\mathsf{role} = \mathsf{visitor}) \wedge (8 \leq \mathsf{time} \leq 20)$. This local policy grants all access requests that assign the value visitor to the attribute role and a number between 8 and 20 to the attribute time. Note that this local policy is also the target of requirement **R1**.

B. Decidability

To show that the policy synthesis problem is decidable, we give a synthesis algorithm, called $\mathcal{S}_{\mathsf{cs}}$, that uses controller synthesis as a subroutine. In the following, we first define the controller synthesis problem. We then show how the algorithm

$\mathcal{S}_{\mathsf{cs}}$ constructs the PEPs' local policies by solving multiple controller synthesis instances.

Controller Synthesis Problem. Controller synthesis algorithms take as input a description of an uncontrolled system, called a plant, along with a specification, and output a controller that restricts the plant so that it satisfies the given specification. In our setting, the plant is the resource structure and the specification is an access constraint, i.e. a CTL formula over resource attributes. The synthesized controller then defines which PEPs must grant or deny the access request so that the access constraint is satisfied. For simplicity, we do not define the controller synthesis problem in its most general form. For our needs the following simpler definition suffices.

Definition 3. *The* controller synthesis problem *is as follows:*

Input. *A resource structure $S = (\mathcal{R}, E, r_e, L)$ and an access constraint φ.*

Output. *A set $E' \subseteq E$ of edges such that $(\mathcal{R}, E', r_e, L) \models \varphi$, if such an E' exists, and* unsat *otherwise.*

The controller synthesis problem can be reduced to synthesizing a memoryless controller for a Kripke structure given a CTL specification. Deciding whether a controller synthesis instance has a solution is NP-complete [15]. Systems such as MBP [16] can be used to synthesize controllers. For a comprehensive overview of controller synthesis see [17].

Algorithm. The algorithm $\mathcal{S}_{\mathsf{cs}}$ is based on two insights. First, for a given access request q, we can use controller synthesis to identify which PEPs must grant or deny q. In more detail, we can compute $(\mathcal{R}, E', r_e, L) \models \varphi_q$, where φ_q conjoins all access constraints of the requirements that are applicable to q. The edges in E' represent the PEPs that must grant q and those in $E \setminus E'$ the PEPs that must deny q. A configuration can thus be synthesized by solving one controller synthesis instance for each access request. However, there are infinitely many access requests. Our second insight is that we can construct a configuration by solving finitely many controller synthesis instances. We partition the set $\mathcal{Q}$ of access requests into $2^{|R|}$ equivalence classes, where two access requests are equivalent if the same set of requirements are applicable to them. Solving one controller synthesis instance for one representative access request per equivalence class is sufficient for our purpose.

The main steps of the algorithm $\mathcal{S}_{\mathsf{cs}}$ are given in Algorithm 1. The algorithm iteratively constructs a configuration c as follows. Initially, it sets all local policies to true (lines 2-3). The algorithm iterates over all subsets $R' = \{T_1 \Rightarrow \varphi_1, \dots, T_i \Rightarrow \varphi_i\}$ of the requirements R (line 4). The conjunction $T = T_1 \wedge \cdots \wedge T_i \wedge \neg T_{i+1} \wedge \cdots \neg T_n$ constructed at line 5 is satisfied by all access requests to which only the requirements contained in R' are applicable. The set $\{q \in \mathcal{Q} \mid q \vdash T\}$ is an equivalence class of access requests. If this equivalence class is nonempty, i.e. $\exists q \in \mathcal{Q}.\ q \vdash T$, then c must grant and deny all access requests contained in it in conformance with the access constraints defined by R'. Lines 9-14 define how the algorithm $\mathcal{S}_{\mathsf{cs}}$ updates c. First, it constructs the conjunction φ of the access constraints defined by the requirement in R'.

Algorithm 1: The algorithm $\mathcal{S}_{\mathsf{cs}}$ for synthesizing policies using controller synthesis. The controller synthesis algorithm, denoted $\mathsf{cs}(S, \varphi)$, outputs either a subset of E or unsat.

```
Input: Resource stricture $S = (\mathcal{R}, E, r_e, L)$,
       a set of requirements $R$
Output: A configuration $c$ or unsat
1  begin
2      for $e \in E$ do
3          $c(e) \leftarrow \mathsf{true}$
4      for $R' \subseteq R$ do
5          $T \leftarrow T_1 \wedge \cdots \wedge T_i \wedge \neg T_{i+1} \wedge \cdots \wedge \neg T_n$, where
6              $\{T_1 \Rightarrow \varphi_1, \dots, T_i \Rightarrow \varphi_i\} = R'$ and
7              $\{T_{i+1} \Rightarrow \varphi_{i+1}, \dots, T_n \Rightarrow \varphi_n\} = R \setminus R'$
8          if $\exists q \in \mathcal{Q}.\ q \vdash T$ then
9              $\varphi \leftarrow \varphi_1 \wedge \cdots \wedge \varphi_i$
10             if $\mathsf{cs}(S, \varphi) = \mathsf{unsat}$ then
11                 return unsat
12             else
13                 for $e \in E \setminus \mathsf{cs}(S, \varphi)$ do
14                     $c(e) \leftarrow c(e) \wedge (\neg T)$
15     return $c$
```

It then executes the controller synthesis algorithm, denoted by cs, with the inputs S and φ. If the algorithm cs returns unsat, then the requirements are not satisfiable for the given resource structure, and the algorithm $\mathcal{S}_{\mathsf{cs}}$ thus returns unsat. Otherwise, the algorithm cs returns a set $E' \subseteq E$ of edges. The algorithm updates the configuration c as follows: for any edge in $E \setminus E'$, the configuration is modified to deny access to all requests in the equivalence class defined by R'. The algorithm terminates when all subsets of the global requirements have been considered.

Theorem 1. *Let S be a resource structure and R a set of requirements. If $\mathcal{S}_{\mathsf{cs}}(S, R) = c$ then $S, c \Vdash R$. If $\mathcal{S}_{\mathsf{cs}}(S, R) =$* unsat *then there is no configuration c such that $S, c \Vdash R$.*

We prove this theorem and give the complexity of $\mathcal{S}_{\mathsf{cs}}$ in [5].

Example. To illustrate $\mathcal{S}_{\mathsf{cs}}$, consider our running example and the requirements **R2** and **R5** formalized as follows:

$$\mathbf{R2} := (\mathsf{role} = \mathsf{visitor}) \Rightarrow (\mathsf{A}[(\mathsf{id} = \mathsf{lob})\ \mathsf{R}\ (\mathsf{id} = \mathsf{mr})])$$
$$\mathbf{R5} := (\mathsf{role} \neq \mathsf{employee}) \Rightarrow (\mathsf{AG}\ \neg\,\mathsf{sec\text{-}zone})\ .$$

Recall that $\mathsf{dom}(\mathsf{role}) = \{\perp, \mathsf{visitor}, \mathsf{employee}\}$, and hence the targets $\mathsf{role} = \mathsf{visitor}$ and $\mathsf{role} \neq \mathsf{employee}$ are not equivalent.

To synthesize a configuration, the algorithm $\mathcal{S}_{\mathsf{cs}}$ executes the second for-loop four times. Let the selected subset of requirements in the first iteration be $\{\mathbf{R2}, \mathbf{R5}\}$. The conjunction T of the targets is $(\mathsf{role} = \mathsf{visitor}) \wedge (\mathsf{role} \neq \mathsf{employee})$, which is equivalent to $(\mathsf{role} = \mathsf{visitor})$. Hence, T is satisfiable. The access constraint φ (see Algorithm 1, line 9) is then $(\mathsf{A}[(\mathsf{id} = \mathsf{lob})\ \mathsf{R}\ (\mathsf{id} = \mathsf{mr})]) \wedge (\mathsf{AG}\ \neg\,\mathsf{sec\text{-}zone})$. A possible output by the controller synthesis algorithm $\mathsf{cs}(S, \varphi)$ is $E \setminus \{(\mathsf{cor}, \mathsf{bur}), (\mathsf{out}, \mathsf{cor})\}$. The updated configuration c after the first iteration is therefore

$$c(e) = \begin{cases} \mathsf{true} \wedge \mathsf{role} \neq \mathsf{visitor} & \text{if } e = (\mathsf{cor}, \mathsf{bur}) \\ \mathsf{true} \wedge \mathsf{role} \neq \mathsf{visitor} & \text{if } e = (\mathsf{out}, \mathsf{cor}) \\ \mathsf{true} & \text{otherwise} . \end{cases}$$

Suppose the outputs to the remaining three controller synthesis instances are $\mathsf{cs}(S, \varphi_{\{\mathbf{R2}\}}) = E \setminus \{(\mathsf{out}, \mathsf{cor})\}$, $\mathsf{cs}(S, \varphi_{\{\mathbf{R5}\}}) = E \setminus \{(\mathsf{cor}, \mathsf{bur})\}$, and $\mathsf{cs}(S, \varphi_{\emptyset}) = E$, where φ_X denotes the conjunction of the access constraints of the requirements in X. The simplified configuration c returned by $\mathcal{S}_{\mathsf{cs}}$ is

$$c(e) = \begin{cases} \mathsf{role} = \mathsf{employee} & \text{if } e = (\mathsf{cor}, \mathsf{bur}) \\ \mathsf{role} \neq \mathsf{visitor} & \text{if } e = (\mathsf{out}, \mathsf{cor}) \\ \mathsf{true} & \text{otherwise} . \end{cases}$$

Limitations. The main limitation of the algorithm $\mathcal{S}_{\mathsf{cs}}$ is that the running time is exponential in the number of requirements, rendering it impractical for nontrivial instances of policy synthesis. For example, while the algorithm $\mathcal{S}_{\mathsf{cs}}$ takes 2 seconds to synthesize a configuration for our running example, it does not terminate within an hour for our case studies, reported in Section VII-B. We give a practical policy synthesis algorithm based on SMT solving in Section VI.

C. *NP-hardness*

To show NP-hardness, we reduce propositional satisfiability to the policy synthesis problem. It is easy to see that a propositional formula φ can be encoded, in logarithmic space, as a target T_φ over Boolean attributes. Consider the policy synthesis problem for the inputs S and $\{(T_\varphi \Rightarrow \mathsf{false})\}$, where S is an arbitrary resource structure. If the output to this policy synthesis instance is unsat then for some access request q, we have $q \vdash T_\varphi$. Hence φ is satisfiable. Alternatively, the output to the policy synthesis problem is a configuration c. Since for any access request q where $q \vdash T_\varphi$ we have $S_{c,q} \models \mathsf{false}$, it is immediate that there is no access request q such that $q \vdash T_\varphi$. Therefore, φ is unsatisfiable.

VI. Policy Synthesis Algorithm

In this section, we define our policy synthesis algorithm based on SMT solving, called $\mathcal{S}_{\mathsf{smt}}$. The algorithm takes as input a resource structure S, a set R of requirements, and a set C of configurations. The set C is encoded symbolically, as we describe shortly. The algorithm outputs a configuration c such that $S, c \Vdash R$, if there is such a configuration in C; otherwise, it returns unsat. To synthesize a configuration c, the algorithm encodes the question $\exists c \in C.\ S, c \Vdash R$ in a decidable logic supported by standard SMT solvers. Due to its technical nature, we relegate a detailed description of the encoding to the end of this section.

Our algorithm takes as input a set of configurations, and we refer to the symbolic encoding of this set as a *configuration template*. The configuration template enables us to restrict the search space: the algorithm confines its search to the configurations described by the template. Our algorithm $\mathcal{S}_{\mathsf{smt}}$ is sound, independent of the provided configuration template. Its completeness, however, depends on the template. We show that one can construct a template for which $\mathcal{S}_{\mathsf{smt}}$ is complete, but the resulting template would, in practice, encode so many configurations that the resulting SMT problem would be infeasible to solve. We therefore strike a balance between the algorithm's completeness and its efficiency: since real-world local policies often have small syntactic representations, as demonstrated by our experiments in Section VII, our policy synthesis tool starts with a template that defines configurations with succinct local policies, and iteratively executes $\mathcal{S}_{\mathsf{smt}}$, increasing the template's size in each iteration. It turns out that in our case studies a small number of iterations is sufficient to synthesize all local policies. Below, we describe the algorithm $\mathcal{S}_{\mathsf{smt}}$'s components.

A. *Configuration Templates*

A *configuration template* assigns to each edge of the resource structure a symbolic encoding of a set of local policies. To illustrate this encoding, consider the set of local policies $\{\mathsf{true}, \mathsf{role} = \mathsf{employee}, \mathsf{role} \neq \mathsf{visitor}\}$. We symbolically encode this set for an edge, say $(\mathsf{cor}, \mathsf{bur})$, as a constraint over subject and contextual attributes, as well as a *control* variable $z_{(\mathsf{cor},\mathsf{bur})}$:

$$\begin{aligned} C((\mathsf{cor}, \mathsf{bur})) = {} & (z_{(\mathsf{cor},\mathsf{bur})} = 1 \Rightarrow \mathsf{true}) \wedge \\ & (z_{(\mathsf{cor},\mathsf{bur})} = 2 \Rightarrow \mathsf{role} = \mathsf{employee}) \wedge \\ & (z_{(\mathsf{cor},\mathsf{bur})} = 3 \Rightarrow \mathsf{role} \neq \mathsf{visitor}) . \end{aligned} \quad \text{(T1)}$$

The control variable $z_{(\mathsf{cor},\mathsf{bur})}$ encodes the choice of one of three local policies for the edge $(\mathsf{cor}, \mathsf{bur})$. Hence, for this example, the set of configurations defined by the configuration template contains $3^{|E|}$ elements, where E is the set of edges in the resource structure. Note that for a set of local policies of size n (here $n = 3$), $\lceil \log n \rceil$ propositional variables are sufficient for representing each edge's control variables. To avoid clutter, we will write C_{r_0,r_1} for $C((r_0, r_1))$.

We remark that configuration templates can be used to restrict the search space of configurations to those that satisfy *attribute availability* constraints, which restrict the set of attributes that PEPs can retrieve. Suppose that only the side-entrance door of our running example is equipped with a keypad. To account for this constraint, we will restrict the configurations in the template to those that use the $\mathsf{correct\text{-}pin}$ attribute only in the local policy of side entrance's lock.

B. *Algorithm*

The main steps of $\mathcal{S}_{\mathsf{smt}}$ are given in Algorithm 2. We describe the algorithm with an example: the input to the algorithm consists of the resource structure and the requirements $\mathbf{R2}$ and $\mathbf{R5}$ of our running example, along with the above configuration template C, which maps edges to the set of local policies $\{\mathsf{true}, \mathsf{role} = \mathsf{employee}, \mathsf{role} \neq \mathsf{visitor}\}$. The algorithm starts by creating for each requirement a constraint that asserts the satisfaction of the requirement in the resource structure, given the template. This constraint is called ψ in the algorithm, and is expressed in the logic of an SMT solver. This step is implemented by the subroutine Encode, defined in Figure 8. To encode the satisfaction of access constraints,

Algorithm 2: The algorithm $\mathcal{S}_{\mathsf{smt}}$ for synthesizing policies using SMT solving.

```
Input: A resource structure $S = (\mathcal{R}, E, r, L)$,
       a set $\{R_1, \cdots, R_n\}$ of requirements,
       a configuration template $C$
Output: A configuration $c$ or unsat
1  begin
2      $\phi \leftarrow \mathsf{true}$
3      for $R \in \{R_1, \cdots, R_n\}$ do
4          $\psi \leftarrow \textsc{Encode}(S, R, C)$
5          $\phi \leftarrow \phi \wedge \psi$
6      if $(\exists \vec{z}.\forall \vec{a}.\ \phi)$ is sat then
7          $\mathcal{M} \leftarrow \textsc{Model}(\exists \vec{z}.\forall \vec{a}.\ \phi)$
8          for $e \in E$ do
9              $c(e) \leftarrow \textsc{Derive}(C(e), \mathcal{M})$
10         return $c$
11     else
12         return unsat
```

we follow the standard model-checking algorithm for CTL based on labeling [18]; we explain this encoding at the end of this section.

As an example, the result of $\textsc{Encode}(S, \mathbf{R2}, C)$, after straightforward simplifications, is the following constraint:

$$\psi_{R2} := \mathsf{role} = \mathsf{visitor} \Rightarrow (\neg C_{\mathsf{out,cor}} \vee \neg C_{\mathsf{cor,mr}})\,.$$

Here role is an *attribute variable*, originating from **R2**'s target, and $C_{\mathsf{out,cor}}$ and $C_{\mathsf{cor,mr}}$ are the symbolic encodings of the local policies for the edges $(\mathsf{out}, \mathsf{cor})$ and $(\mathsf{cor}, \mathsf{mr})$, respectively. This constraint states that if the requirement's target $\mathsf{role} = \mathsf{visitor}$ is satisfied, then one of the PEPs along the path that starts at the entry resource and reaches the meeting room directly through the corridor must deny access. Similarly, $\textsc{Encode}(S, \mathbf{R5}, C)$ returns the constraint:

$$\psi_{R5} := \mathsf{role} \neq \mathsf{employee} \Rightarrow \\ ((\neg C_{\mathsf{out,cor}} \vee \neg C_{\mathsf{cor,bur}}) \\ \wedge (\neg C_{\mathsf{out,lob}} \vee \neg C_{\mathsf{lob,cor}} \vee \neg C_{\mathsf{cor,bur}}))\,.$$

This states that any access request that maps the attribute role to a value other than employee must be denied by at least one PEP along the path to the bureau that goes directly through the corridor, and moreover it must be denied by at least one PEP along the path that passes through the lobby.

The conjunction of the constraints created for all the requirements is called ϕ in Algorithm 2. To check whether there is a configuration in C that satisfies the requirements, the algorithm calls an SMT solver to find a model for the formula $\exists \vec{z}.\forall \vec{a}.\ \phi$. Here this is

$$\exists \vec{z}.\forall \vec{a}.\ (\psi_{R2} \wedge \psi_{R5})\ ,$$

where $\vec{z}$ and $\vec{a}$ consist, respectively, of all the control and attribute variables. If ϕ is unsatisfiable, then no configuration in C satisfies the requirements. In this case, the algorithm returns unsat. If however the formula is satisfiable, then the SMT solver returns a model of the formula, which instantiates all the control variables (but not the attribute variables since they are universally quantified). We refer to the SMT solver's procedure that returns such a model as $\textsc{Model}$ in Algorithm 2. The model $\mathcal{M}$ generated by the SMT solver in effect identifies the local policy for each edge e: by instantiating the control variables in $C(e)$, we obtain e's local policy; see template T1. This procedure is called $\textsc{Derive}(C(e), \mathcal{M})$ in the algorithm. For our example, a model $\mathcal{M}$ that satisfies $\exists \vec{z}.\forall \vec{a}.\ (\psi_{R2} \wedge \psi_{R5})$ maps $z_{(\mathsf{cor,bur})}$ to 2, $z_{(\mathsf{out,cor})}$ to 3, and all other control variables to 1. It is then evident from template T1 that, e.g., the local policy for the edge $(\mathsf{cor}, \mathsf{bur})$ is $(\mathsf{role} = \mathsf{employee})$.

$\textsc{Encode}(S, T \Rightarrow \varphi, C)$ **returns** $T \Rightarrow \tau(\varphi, r_e)$

Rewrite rules $\tau(\varphi, r_0)$:

$$\tau(\mathsf{true}, r_0) \hookrightarrow \mathsf{true}$$

$$\tau(a \in D, r_0) \hookrightarrow \begin{cases} \mathsf{true} & \text{if } L(r_0)(a) \in D \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\tau(\neg\varphi, r_0) \hookrightarrow \neg\tau(\varphi, r_0)$$

$$\tau(\varphi_1 \wedge \varphi_2, r_0) \hookrightarrow \tau(\varphi_1, r_0) \wedge \tau(\varphi_2, r_0)$$

$$\tau(\mathsf{EX}\varphi, r_0) \hookrightarrow \exists r_1 \in E(r_0).\ \big(C_{r_0,r_1} \wedge \tau(\varphi, r_1)\big)$$

$$\tau(\mathsf{AX}\varphi, r_0) \hookrightarrow \forall r_1 \in E(r_0).\ \big(C_{r_0,r_1} \Rightarrow \tau(\varphi, r_1)\big)$$

$$\tau(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0) \hookrightarrow \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset)$$

$$\tau(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0) \hookrightarrow \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, \emptyset)$$

Rewrite rules $\tau_{\mathsf{U}}(\varphi, r_0, X)$, with $X \subseteq \mathcal{R}$:

$$\tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_0, X) \hookrightarrow \tau(\varphi_2, r_0) \vee \Big(\tau(\varphi_1, r_0) \wedge \\ \big(\exists r_1 \in E(r_0) \backslash X.\ C_{r_0,r_1} \wedge \tau_{\mathsf{U}}(\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})\big)\Big)$$

$$\tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_0, X) \hookrightarrow \tau(\varphi_2, r_0) \vee \Big(\tau(\varphi_1, r_0) \wedge \\ \big(\forall r_1 \in E(r_0) \backslash X.\ C_{r_0,r_1} \Rightarrow \tau_{\mathsf{U}}(\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})\big) \wedge \\ \big(\forall r_1 \in E(r_0) \cap X.\ \neg C_{r_0,r_1}\big)\Big)$$

Fig. 8: Encoding the satisfaction of a requirement $T \Rightarrow \varphi$ in a resource structure $S = (\mathcal{R}, E, r_e, L)$, given a template C, into an SMT constraint. The rewrite rules τ reduce an access constraint φ and a resource r_0 to an SMT constraint. For a resource $r_0 \in \mathcal{R}$, we write $E(r_0)$ for $\{r_1 \in \mathcal{R} \mid (r_0, r_1) \in E\}$. The $\exists$ and $\forall$ quantifiers range over a finite domain. Therefore, the former can be expanded as a finite number of disjunctions, and the latter as a finite number of conjunctions.

Complexity. Let S be a resource structure, R be a set of requirements, and C be configuration template. The running time of the $\mathcal{S}_{\mathsf{smt}}$ algorithm is determined by the size of the generated formula ϕ and the complexity of finding a model of ϕ. The size of the formula ϕ is in $\mathcal{O}(d \cdot |R| \cdot |\mathcal{R}|)$, where d is the size of the largest access constraint that appears in the requirements, R is the set of requirements, and $\mathcal{R}$ is the set of resources in S. The formula ϕ is defined over Boolean control variables $\vec{z}$ and attribute variables $\vec{a}$. The number of control and attribute variables is $\lceil log(|C|)\rceil$ and $|\mathcal{A}|$, respectively. In the worst case, one must check all possible models of the formula ϕ, so finding a model of ϕ is in $\mathcal{O}(2^{\lceil log(|C|)\rceil + k\cdot|A|})$, where k is the largest domain that appears in the constraints. Note that such domains are always finite. For example, $\mathsf{time} \geq 10$ is a shorthand for $\neg(\mathsf{time} \in \{0,\ldots,9\})$. We conclude that the overall running time of the algorithm $\mathcal{S}_{\mathsf{smt}}$ is in $\mathcal{O}(2^{\lceil log(|C|)\rceil + k\cdot|A|} + d\cdot|R|\cdot|\mathcal{R}|)$.

C. Soundness and Completeness

The algorithm $\mathcal{S}_{\mathsf{smt}}$ is sound.

Theorem 2. *Let S be resource structure, R a set of requirements, and C a configuration template. If $\mathcal{S}_{\mathsf{smt}}(S,R,C) = c$ then $S,c \Vdash R$. If $\mathcal{S}_{\mathsf{smt}}(S,R,C) = \mathsf{unsat}$, then there is no configuration c in C such that $S,c \Vdash R$.*

$\mathcal{S}_{\mathsf{smt}}$'s completeness depends on the template C provided as input to the algorithm. We show that one can construct a template for which the algorithm is complete. A template C is *complete* for a given resource structure S and set of requirements R if $\mathcal{S}_{\mathsf{smt}}(S,R,C)$ returns a configuration whenever there is a configuration that satisfies the requirements. For the algorithm's completeness, it is in fact sufficient to start the algorithm with a template $C_{S,R}$ that contains all the configurations that the algorithm based on controller synthesis, described in Section V-B, may output. The following theorem formalizes this observation.

Theorem 3. *Given a resource structure S and a set R of requirements, the configuration template $C_{S,R}$ is complete for S and R.*

The number of configurations in $C_{S,R}$ is exponential in $|E|$ and $|R|$ (which we prove in [5]). Hence this template, although complete, is not useful in practice as it would overwhelm SMT solvers, rendering $\mathcal{S}_{\mathsf{smt}}$ ineffective. In Section VII-A, where we explain our implementation in detail, we describe a configuration template that works well for synthesizing configurations for practically-relevant examples.

We conclude this discussion by pointing out that our synthesis algorithm can be readily used to verify whether a candidate configuration c satisfies a set R of global access-control requirements in a resource structure S. Namely, if the configuration template input to $\mathcal{S}_{\mathsf{smt}}$ consists only of the configuration c, then $\mathcal{S}_{\mathsf{smt}}$ returns c if $S,c \models R$; otherwise, the algorithm returns unsat, which means that the configuration c does not satisfy R.

D. Encoding into SMT

We now explain Algorithm 2's procedure ENCODE, which translates a resource structure S, a requirement $R = (T \Rightarrow \varphi)$, and a configuration template C, into an SMT constraint $T \Rightarrow \tau(\varphi, r_e)$. The generated constraint encodes that whenever the requirement $T \Rightarrow \varphi$ is applicable to an access request q, i.e. $q \vdash T$, then φ must be satisfied for the entry resource r_e in the structure $S_{c,q}$. Here, c is the configuration selected from the template C. The constraint $\tau(\varphi, r_e)$ is generated using the rewrite rules τ as defined in Figure 8.

Given an access constraint φ and a resource r_0, the rewrite rules τ produce an SMT constraint $\tau(\varphi, r_0)$ that encodes $S, r_0 \models \varphi$; see Figure 6. The rewrite rules for access constraints of the form true, $a \in D$, $\neg\varphi$, and $\varphi_1 \wedge \varphi_2$ are as expected. The rewrite rule for access constraints of the form $\mathsf{EX}\varphi$ encodes that the access constraint φ is satisfied at r_0 if there is an edge from r_0 to some node r_1 such that C_{r_0,r_1} holds and $S, r_1 \models \varphi$. In this rule, the constraint C_{r_0,r_1} returns the symbolic encoding of the local policies for the edge (r_0, r_1), and $\tau(\varphi, r_1)$ returns the encoding of $S, r_1 \models \varphi$ as an SMT constraint. In contrast to EX, the rewrite rule for $\mathsf{AX}\varphi$ access constraints states that for any resource r_1, such that $(r_0, r_1) \in E$, if C_{r_0,r_1} is true then the constraint $\tau(\varphi, r_1)$ is satisfied.

To encode the semantics of the connectives EU (AU), we use the *until* rewrite rules τ_{U}, which reduce an until construct $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ ($\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$), a resource $r_0 \in \mathcal{R}$, and a set of resources $X \subseteq \mathcal{R}$ to an SMT constraint. We use the set of resources X to record for which resources the satisfaction of the until access constraint has already been encoded. This is necessary to guarantee the reduction system's termination. The rule for access constraints of the form $\mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$ encodes that either $S, r_0 \models \varphi_2$, or $S, r_0 \models \varphi_1$ and there is an edge from r_0 to some node r_1 such that C_{r_0,r_1} holds and $S, r_1 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Here $\tau_{\mathsf{U}}(E[\varphi_1 \mathsf{U} \varphi_2], r_1, X \cup \{r_0\})$ returns the encoding of $S, r_1 \models \mathsf{E}[\varphi_1 \mathsf{U} \varphi_2]$. Note that we add r_0 to X to ensure that no resource is revisited during EU-rewriting. Similarly, the rule for access constraints $\mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ encodes that either $S, r_0 \models \varphi_2$, or $S, r_0 \models \varphi_1$ holds, for any outgoing edge to a node r_1 we have $S, r_1 \models \mathsf{A}[\varphi_1 \mathsf{U} \varphi_2]$ and it has no outgoing edges to nodes in X.

We illustrate our encoding with examples in [5]. There, we also prove that this rewrite system always terminates, and that the generated SMT encoding of access constraints is correct.

VII. Implementation and Evaluation

We report on an implementation of our policy synthesis algorithm, the case studies we conducted to evaluate its efficiency and scalability, and our empirical results.

A. Implementation

We have implemented a synthesizer that encodes policy synthesis instances into the QF_LIA and QF_UA logics of SMT-LIB v2 [19] and uses the Z3 SMT solver [20]. Our synthesizer is configured with configuration templates of different sizes. The local policies defined by these configuration templates are in disjunctive normal form. Namely, the local policies are defined as a disjunction of clauses, each clause consisting of a conjunction of terms, where each term is either an equality constraint for non-numerical attributes (e.g. $\mathsf{role} = \mathsf{employee}$) or an interval constraint for numeric attributes (e.g. $t_1 \leq \mathsf{time} \leq t_2$). We denote by C_k the configuration template that defines local policies with k clauses, each consisting of k

terms. Note that the local policies defined in the template C_k may refer to at most k^2 attributes.

Our synthesizer implements the following procedure: it iteratively executes $\mathcal{S}_{\mathsf{smt}}(S, R, C_1)$, $\mathcal{S}_{\mathsf{smt}}(S, R, C_2)$, $\mathcal{S}_{\mathsf{smt}}(S, R, C_3), \ldots$, stopping with the first call to $\mathcal{S}_{\mathsf{smt}}$ that returns a satisfying configuration, and returning this configuration. By iterating over templates increasing in size, our synthesizer generates small local policies, which is desirable for avoiding redundant attribute checks. For the running example, for instance, our synthesizer's output includes the constraint $\mathsf{correct\text{-}pin}$ only for the entrance gates' local policies, and does not include this check, e.g., for the office room's policy. A satisfying solution for each case study can be found in the configuration template C_3. This indicates that real-world local policies have concise representations.

Note that our synthesizer may not terminate in a reasonable amount of time if no configuration satisfies the global requirements for the given resource structure. In our case studies, we used a simple iterative method to pinpoint such unsatisfiable requirements: we start with a singleton set of requirements, consisting of one satisfiable requirement, and iteratively extend this set by one requirement. This helped us identify a minimal set of conflicting requirements and revise problematic ones.

B. Case Studies

To investigate $\mathcal{S}_{\mathsf{smt}}$'s efficiency and scalability, we have conducted case studies in collaboration with KABA. We used real-world requirements and resource structures, and used our tool to synthesize policy configurations for a university building, a corporate building, and an airport terminal. Our synthesizer and all data are publicly available[1]. Below, we briefly explain the three case studies; relevant complexity metrics are summarized in Table I.

University Building. We modeled the main floor of ETH Zurich's computer science building. This floor consists of 66 subspaces including labs, offices, meeting rooms, and shared areas. The subspaces are labeled with four attributes that indicate: the research group to which a physical space is assigned, the physical space type (e.g., office, teaching room, or server room), the room number, and whether the physical spaces belongs to a secretary or a faculty member. Example requirements stipulate that a research group's PhD students can access all offices assigned to the group except those assigned to the faculty members and secretaries. The policies are defined over eight attributes.

Corporate Building. We modeled an office space that consists of 20 subspaces, including a lobby, meeting rooms, offices, and restricted areas such as a server room, a mail room, and an HR office. The rooms are connected by three corridors, and they are labeled with attributes to mark public areas and employee-only zones. Access to these spaces is controlled by locks that are equipped with smartcard readers and PIN keypads. These locks are connected to a time server. Example requirements are that only the postman and HR employees can access the mail room, and that between noon and 1PM employees can access their offices without entering their PIN. The policies are defined over four attributes.

[1]http://www.infsec.ethz.ch/research/software/spctl.html

		University building	Corporate building	Airport terminal
Complexity metrics	Requirements	14	10	15
	PEPs	127	41	32
	Subspaces	66	20	13
Performance	Synthesis time	10.32	25.30	1.92
	Std. dev.	0.04	0.15	0.01

TABLE I: Complexity metrics and policy synthesis times (in seconds) for the three cases studies

Airport Terminal. We modeled the main terminal of a major international airport. The part of the terminal that we modeled includes subspaces such as the boarding pass control, security, and shopping areas. We have used the actual plan of the terminal, and considered 15 requirements, all currently enforced by the airport's access-control system. The area is divided into 13 subspaces, each labeled with zone identifiers (such as check-in and passport control). Example requirements stipulate that no passenger can access departure areas before passing through security, passengers with economy boarding passes cannot pass through the business/first-class ticket-control gates, and that only airport staff can access certain elevators.

C. Empirical Results

We ran all experiments on a Linux machine with a quad-code i7-4770 CPU, 32GB of RAM, running Z3 SMT v4.4.0. We present two sets of results: (1) the synthesizer's performance when used to synthesize the local policies for the three case studies, and (2) the synthesizer's scalability.

Performance. We used our tool to synthesize the local policies for the three case studies, measuring the time taken for policy synthesis. We report the average synthesis time, measured over 10 runs of the synthesizer, in the bottom two rows of Table I. The reported synthesis time is the sum of the time taken for encoding the policy synthesis instance into SMT constraints, the time for solving the generated SMT constraints, and the time for iterating over the smaller templates for which the synthesizer returns unsat. In all three case studies, our tool synthesizes the local policies in less than 30 seconds. The standard deviation is under 0.2 seconds. This indicates that synthesizing local policies is practical, and can be used for real-world systems.

Scalability Experiments. To investigate the scalability of our synthesis tool, we synthetically generated larger problem instances based on the corporate building case study. Although the case study originally consisted of a single floor, we increased the number of the floors in the building. We kept the same labeling for the newly added subspaces, so the original requirements also pertain to the newly added floors. Based on this method, we scaled the number of PEPs up to 670.

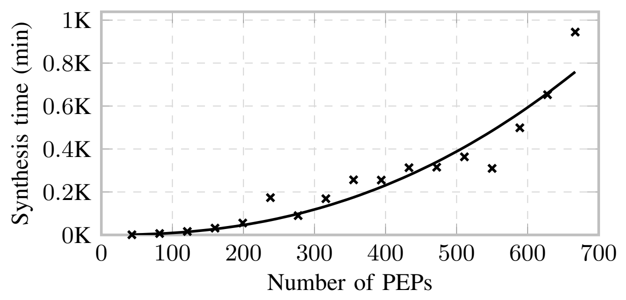


Fig. 9: Scaling the number of PEPs

The time needed to synthesize local policies for different numbers of PEPs is given in Figure 9. The results show that our tool can synthesize a large number of local policies in a reasonable amount of time. For example, synthesizing up to 600 local policies takes less than ten hours. The tool's performance can be further improved using domain-specific heuristics for solving the resulting SMT constraints. Nevertheless, the tool already scales to most real-world scenarios: protected physical spaces usually have less than 500 PEPs.

VIII. Related Work

Physical Access Control. The Grey project was an experiment in deploying a physical access-control system at the campus of Carnegie Mellon University [21], [22]. As part of this project, researchers developed formal languages for specifying policies and credentials, and also developed techniques for detecting policy misconfigurations [3], [23]. The work on credential management, such as delegation, is orthogonal to the specification of the locks' local policies. In contrast to their work on detecting policy misconfigurations, we have developed a framework to synthesize policies that are guaranteed to enforces the global requirements, avoiding misconfigurations.

Several researchers have investigated SAT-based and model-checking techniques for reasoning about physical access control [1], [2]. Similarly to our work, these approaches model spatial constraints, and formalize global requirements that physical access-control systems must enforce. The authors of [1], for instance, model physical spaces using directed graphs and formalize global requirements in first-order logic. Their goal is to identify undesired denials due to blocked paths and unintended grants to restricted zones using SAT solvers. In contrast to these verification approaches, we develop a synthesis framework for generating correct local policies.

Network Policy Synthesis. The problems of configuring networks with access-control and routing policies are related to the problem of constructing local policies from global requirements. In the network problem domain, one has an explicit resource structure defined by the network topology and must enforce global requirements using local rules deployed at the switches. Several synthesis algorithms for networks have been studied; e.g. see [24]–[30]. The authors of [24] and [25], for example, propose techniques for synthesizing local firewall rules that collectively enforce global network requirements in a given network topology. These approaches are sufficiently expressive for formalizing simple connectivity constraints, such as which hosts can access which services in a network. Similarly to our approach, recent techniques for synthesizing network configurations, such as [26]–[29], also leverage SAT and SMT solvers. In addition to access-control constraints, these techniques also consider business constraints, such as deployment cost and usability. However, none of the above approaches for network synthesis supports branching properties, which are necessary for specifying requirements such as those stipulating that a fire-exit is reachable from any office room, as well as those that instantiate our waypointing and blocking requirement patterns; see Section IV-B for examples. These requirements, which can be expressed in our framework, are central to physical access control. Existing network policy synthesis algorithms, therefore, are not sufficiently expressive for handling access-control requirements for physical spaces.

Policy verification has also been studied in the context of computer networks; see e.g. [31]. However, this line of research is not concerned with synthesis, which is our work's main focus. We remark though that our synthesis algorithm can be readily used for verifying the conformance of a set of local policies to global access-control requirements; see Section VI.

Program Synthesis. Program synthesis techniques, such as template-based synthesis [32]–[35], reactive program synthesis from temporal specifications [36]–[38], and program repair techniques [39], [40], are related to policy synthesis for physical spaces. Similarly to our SMT-based algorithm, most of these synthesis frameworks also supplement the logical specification with a template, and exploit SMT solvers to efficiently explore the search space defined by the template. They cannot however express the relevant access-control requirements we have considered, such as those pertaining to branching properties. Our synthesis framework builds upon these techniques, and extends them with support for specifications that are needed for physical spaces.

Methods for synthesizing models of logical formulas, such as those in linear-temporal logic or CTL, have been extensively studied in the literature [17], [36], [41]–[44]. In Section V, we have described a policy synthesis algorithm based on CTL controller synthesis. This algorithm however comes at the expense of an exponential blow-up. Therefore, existing CTL synthesis tools and algorithms cannot be readily applied to synthesize attribute-based local policies in practice. Our efficient SMT-based algorithm addresses this practical challenge.

IX. Conclusion

We have presented a framework for synthesizing locally enforceable policies from global, system-wide access-control requirements for physical spaces. Its key components are (1) a declarative language along with patterns for writing global requirements, (2) a model of the physical space describing how subjects access resources, and (3) an efficient policy synthesis algorithm for generating policies compliant with the requirements and the spatial constraints. Using real-world case studies, we have demonstrated that our synthesis framework is practical and scales to systems with complex requirements and numerous policy enforcement points.

As future work, we plan to extend our policy synthesis framework with architectural constraints. Examples include *optimality* constraints, which can be used to synthesize local policies that avoid re-checking attributes that have been checked by other enforcement points. Handling such constraints is important for large-scale access-control systems in practice. We also plan to apply our framework to synthesize locally enforceable policies in other access-control domains, such as access control in networks, which may require tailored synthesis heuristics.

REFERENCES

[1] W. M. Fitzgerald, F. Turkmen, and S. N. Foley, "Anomaly analysis for physical access control security configuration," in *CRiSIS*. IEEE, 2012, pp. 1–8.

[2] R. Frohardt, B.-Y. Chang, and S. Sankaranarayanan, "Access nets: Modeling access to physical spaces," in *VMCAI*. Springer Berlin Heidelberg, 2011, pp. 184–198.

[3] L. Bauer, Y. Liang, M. K. Reiter, and C. Spensky, "Discovering access-control misconfigurations: New approaches and evaluation methodologies," in *CODASPY*. ACM, 2012, pp. 95–104.

[4] E. Emerson and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, 1982.

[5] P. Tsankov, , M. Torabi Dashti, and D. Basin, "Access control synthesis for physical spaces," Tech. Rep., 2016, http://arxiv.org/abs/1605.01769.

[6] "eXtensible Access Control Markup Language (XACML) Version 3.0."

[7] "Goji lock," http://gojiaccess.com/.

[8] "Augustus: Smart home access products," http://august.com.

[9] "Bolt: Unlock your door without keys," http://lockitron.com.

[10] E. A. Emerson, "Handbook of Theoretical Computer Science," J. van Leeuwen, Ed. MIT Press, 1990, vol. B, ch. Temporal and Modal Logic, pp. 995–1072.

[11] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, pp. 1278–1308, 1975.

[12] J. Crampton and C. Morisset, "PTaCL: A language for attribute-based access control in open systems," in *POST*. Springer Berlin Heidelberg, 2012, pp. 390–409.

[13] P. Tsankov, S. Marinovic, M. Torabi Dashti, and D. Basin, "Decentralized composite access control," in *POST*. Springer Berlin Heidelberg, 2014, pp. 245–264.

[14] ——, "Fail-secure access control," in *CCS*. ACM, 2014, pp. 1157–1168.

[15] M. Antoniotti and B. Mishra, "The supervisor synthesis problem for unrestricted CTL is NP-complete," New York University, Tech. Rep., 1995.

[16] P. Bertoli, A. Cimatti, M. Pistore, M. Roveri, and P. Traverso, "MBP: A model based planner," in *IJCAI'01 Workshop on Planning under Uncertainty and Incomplete Information*, 2001.

[17] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.

[18] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*. New York, NY, USA: Cambridge University Press, 2004.

[19] C. Barrett, A. Stump, C. Tinelli, S. Boehme, D. Cok, D. Deharbe, B. Dutertre, P. Fontaine, V. Ganesh, A. Griggio, J. Grundy, P. Jackson, A. Oliveras, S. Krsti, M. Moskal, L. D. Moura, R. Sebastiani, T. D. Cok, and J. Hoenicke, "The SMT-LIB standard: Version 2.0," 2010.

[20] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*. Springer Berlin Heidelberg, 2008, pp. 337–340.

[21] L. Bauer, S. Garriss, and M. K. Reiter, "Distributed proving in access-control systems," in *S&P*. IEEE, 2005, pp. 81–95.

[22] ——, "Efficient proving for practical distributed access-control systems," in *ESORICS*. Springer Berlin Heidelberg, 2007, pp. 19–37.

[23] ——, "Detecting and resolving policy misconfigurations in access-control systems," *ACM TISSEC*, vol. 14, no. 1, pp. 1–28, 2011.

[24] J. D. Guttman, "Filtering postures: Local enforcement for global policies," in *S&P*. IEEE, 1997, pp. 120–129.

[25] Y. Bartal, A. Mayer, K. Nissim, and A. Wool, "Firmato: a novel firewall management toolkit," in *S&P*. IEEE, 1999, pp. 17–31.

[26] J. McClurg, H. Hojjat, P. Černý, and N. Foster, "Efficient synthesis of network updates," in *PLDI*. ACM, 2015, pp. 196–207.

[27] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, 2008.

[28] B. Zhang and E. Al-Shaer, "On synthesizing distributed firewall configurations considering risk, usability and cost constraints," in *CNSM*, 2011.

[29] M. A. Rahman and E. Al-Shaer, "A formal framework for network security design synthesis." in *ICDCS*. IEEE, 2013, pp. 560–570.

[30] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham, "Decentralizing SDN policies," in *POPL'15*. ACM, 2015, pp. 663–676.

[31] E. Al-Shaer and H. Hamed, "Discovery of policy anomalies in distributed firewalls," in *INFOCOM*. IEEE, 2004, pp. 2605–2616.

[32] A. Solar-Lezama, "Program synthesis by sketching," Ph.D. dissertation, University of California, Berkeley, 2008.

[33] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis," in *POPL*. ACM, 2010, pp. 313–326.

[34] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis," in *FMCAD*, 2013, pp. 1–25.

[35] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster, "Path-based inductive synthesis for program inversion," in *PLDI*. ACM, 2011, pp. 492–503.

[36] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*. Springer Berlin Heidelberg, 1982, pp. 52–71.

[37] B. Jobstmann and R. Bloem, "Optimizations for LTL synthesis," in *FMCAD*, 2006, pp. 117–124.

[38] A. Morgenstern and K. Schneider, "Program sketching via CTL* model checking," in *SPIN Conference on Model Checking Software*. Springer Berlin Heidelberg, 2011, pp. 126–143.

[39] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*. Springer Berlin Heidelberg, 2005, pp. 226–238.

[40] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing model checking in verification by ai techniques," *Journal of Artificial Intelligence*, vol. 112, pp. 57–104, 1999.

[41] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *POPL*. ACM, 1989, pp. 179–190.

[42] A. Gromyko, M. Pistore, and P. Traverso, "Supervisory control via symbolic model checking," Tech. Rep., 2006.

[43] ——, "A tool for controller synthesis via symbolic model checking," in *WODES*. IEEE, 2006, pp. 475–476.

[44] M. Antoniotti, "Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the control-d system," Ph.D. dissertation, NYU, 1995.