Dynamic Access Control in Industry 4.0 Systems

Robert Heinrich
Karlsruhe Institute of Technology, Germany
robert.heinrich@kit.edu

Stephan Seifermann
Karlsruhe Institute of Technology, Germany
stephan.seifermann@kit.edu

Maximilian Walter
Karlsruhe Institute of Technology, Germany
maximilian.walter@kit.edu

Sebastian Hahner
Karlsruhe Institute of Technology, Germany
sebastian.hahner@kit.edu

Ralf Reussner
Karlsruhe Institute of Technology, Germany
ralf.reussner@kit.edu

Tomáš Bureš
Charles University, Prague, Czech Republic
bures@d3s.mff.cuni.cz

Petr Hnětynka
Charles University, Prague, Czech Republic
hnetynka@d3s.mff.cuni.cz

Jan Pacovský
Charles University, Prague, Czech Republic
pacovsky@d3s.mff.cuni.cz

February 2, 2023

Abstract

Industry 4.0 enacts ad-hoc cooperation between machines, humans, and organizations in supply and production chains. The cooperation goes beyond rigid hierarchical process structures and increases the levels of efficiency, customization, and individualisation of end-products. Efficient processing and cooperation requires exploiting various sensor and process data and sharing them across various entities including computer systems, machines, mobile devices, humans, and organisations. Access control is a common security mechanism to control data sharing between involved parties. However, access control to virtual resources is not sufficient in presence of Industry 4.0 because physical access has a considerable effect on the protection of information and systems. In addition, access control mechanisms have to become capable of handling dynamically changing situations arising from ad-hoc horizontal cooperation or changes in the environment of Industry 4.0 systems. Established access control mechanisms do not consider dynamic changes and the combination with physical access control yet. Approaches trying to address these shortcomings exist but often do not consider how to get information such as the sensitivity of exchanged information. This chapter proposes a novel approach to control physical and virtual access tied to the dynamics of custom product engineering, hence, establishing confidentiality in ad-hoc horizontal processes. The approach combines static design-time analyses to discover data properties with a dynamic runtime access control approach that evaluates policies protecting virtual and physical assets. The runtime part uses data properties derived from the static design-time analysis, as well as the environment or system status to decide about access.

1 Introduction

Industry 4.0 combines many different areas such as Digital Manufacturing, Internet of Things, or Cyber-physical Systems [10]. In contrast to classic software systems, the interaction with the real world via sensors and actors is a core feature and enabler for many expected benefits. However, this connection between virtual and physical world also imposes threats. Industrial systems are valuable and frequent subjects to security attacks [25, 11], which can lead to high monetary loss because of stopped production or lost business secrets. Controlling access to resources is one of the most fundamental security requirement [12] that makes attacks more challenging.

In presence of industrial systems, it is not sufficient to focus only on virtual resources such as software systems as part of access control. Physical access control such as limited access to workplaces is crucial as well to protect intellectual property. Employing separate solutions for virtual and physical systems to protect resources is possible. However, combining these access control mechanisms can strengthen security even more or make defining access control policies simpler. For instance, virtual access control can limit access to the user interfaces of a production system, i.e. the software system controlling machines, and physical access control limits access to the production hall in addition, which strengthens the protection. In addition, access control policies become more precise when considering dynamically changing properties of the environment instead of only focusing on subjects and objects. For instance, it is not necessary to give all employees access to the production hall but only those that are assigned to a shift taking place in the very same production hall.

Established access control models [14] usually can express policies in an appropriate way but corresponding mechanisms do not consider dynamically changing properties and provide no means to react to them except for rewriting or at least adjusting policies. Such mechanisms do not scale along with frequent changes. On the other side, dynamic approaches proposed in related work such as the ones discussed in Section 6 do not provide means for exploiting design-time information and therefore leave open the question where to get such information from.

To bridge this gap, we propose an approach to formulate access control policies taking into account dynamically changing properties of the environment, the accessing subject and the accessed object as well as a policy evaluation process during runtime. The policies specify situations involving possibly dynamic properties and provide a reaction to this situation, i.e. access is allowed or denied. During runtime, these policies also consider the sensitivity of data that might be accessed. To determine this sensitivity, we propose a static analysis during design time. The design-time analysis lowers the computational effort during runtime because the results can be gathered before the execution.

We demonstrate our approach by applying it to a realistic scenario covering physical and virtual access control. One dynamic property of the scenario are the locations of factory workers, which influence other properties such as the assignments of workers to shifts. Another property is the status of the worker, i.e. if he/she has collected protective gear that is mandatory for accessing the workspaces. The approach was capable of making appropriate access control decisions for this dynamically changing scenario.

The chapter is organized as follows. Section 2 introduces a running example we will use throughout the chapter. The static data flow analysis to reason about data available in early designs are discussed in Section 3. Access Control in highly dynamic environments is discussed in Section 4. Section 5 describes application scenarios of our approach. Section 6 discusses related work. The chapter concludes with a summary and outlook in Section 7.

2 Running Example

Our running example is an extended Industry 4.0 scenario from [3]. This running example focuses on dynamic physical and virtual access control during a production shift in a factory. Figure 1 illustrates the floor plan of the factory. It consists of a main gate for entering the factory, one dispenser for storing safety gear, three workplaces with gates, and multiple machines within the workplaces. The users in the system are workers and shift supervisors. Each worker and supervisor is assigned a working shift and a working place, where they work. Additionally, a shift also contains information about possible replacement workers in case a worker is unavailable. The access to the factory is granted for each worker about 30 minutes before their assigned shift. Then they are also allowed to open the dispenser to retrieve their safety gear. The access to the workplace is granted only with their safety gear and based on their assigned shift. Within their workplace, they can access sensitive information from the machines such as the precise temperature. For instance, outside their workplace, they only can access aggregated values, containing less sensitive information. Additionally, the supervisors can track the whole process in case of an incident, such as a late worker. In case workers are late for their shift, the supervisor can access the workers' phone numbers and send them notifications. Also, in case some workers have not arrived till 15 minutes before the shift, the system automatically revokes their access rights and selects replacements workers. These replacements are then automatically assigned access rights for the shift and their workplace. Figure 1 shows the scenario before a shift. Two groups of workers are moving to their workplaces. The ones near the main gate are without their safety gear

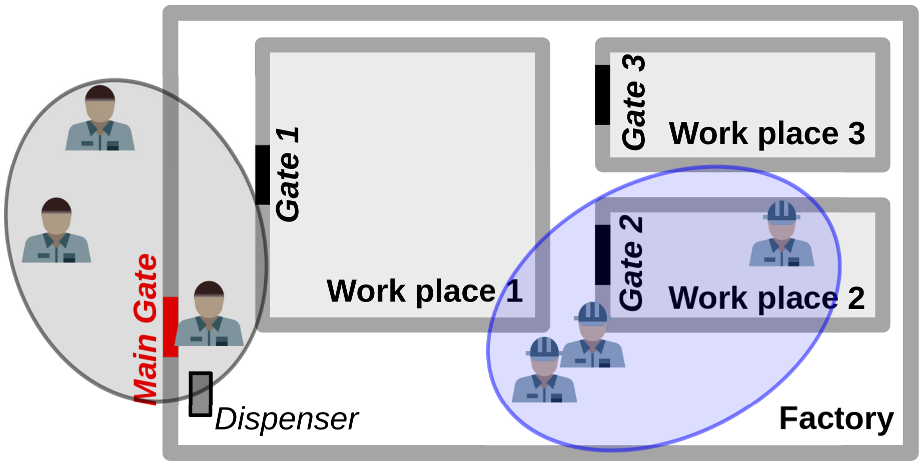


Figure 1: Floor plan of the running example with different workers [3]

and the ones near workplace 2 have already gotten their safety gear.

3 Static Data Flow Analysis

The static data flow analysis provides the dynamic access control system with the sensitivity of data available at all places in the software design. In order to do that, designers have to model the data processing of the system and run a data flow analysis that yields the sensitivity of data. The used modeling and analysis approach [27, 29] is capable of determining other information about processed data as well, but we focus on sensitivity for the sake of simplicity here. In the following, we give an overview on the modeling language and the analysis results. More details on the tooling are available in Section 5.

We use the Palladio modeling language [23] to describe the system design. Modeling the software architecture in Palladio is done by four different roles: component developer, system architect, system deployer, and domain expert. All roles edit their corresponding models in Palladio separately. Figure 2 illustrates the relation between roles and models in Palladio. The component specification stores descriptions of software components and the internal behavior of components. The assembly model stores information about the wiring of component instances. The allocation model stores information about the deployment of component instances. The usage model contains an abstract description of the user behavior and the workload. In general, there is a large amount of modeling languages that we could have used. Especially, there are many languages focused on access control that often also support further security properties (see Section 6 for more details). We have chosen Palladio for several reasons: i) Palladio is a domain specific modeling language for describing software architectures. Therefore, it avoids ambiguities often introduced by generic modeling languages that would require using imprecise heuristics. ii) Palladio supports expressing classic aspects of system structures that can be found in other modeling languages as well. This includes service signatures, components providing services and calls between services. Therefore, we as-

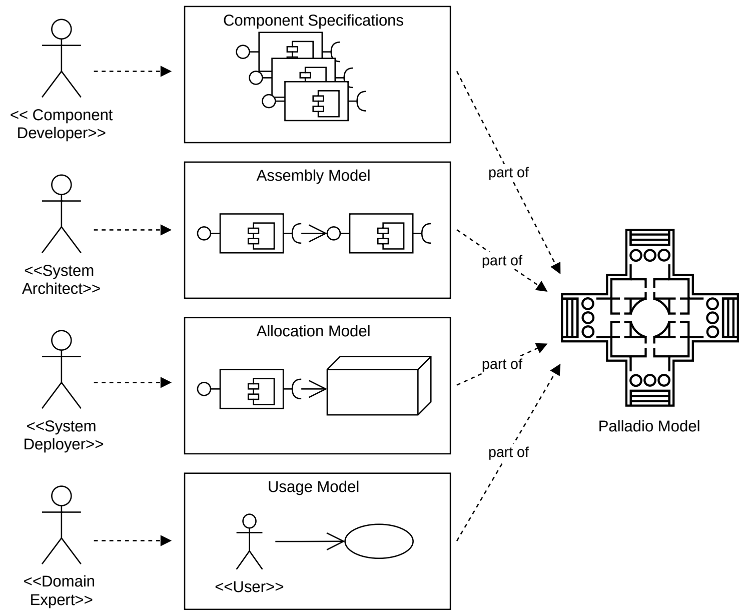


Figure 2: Overview of the classical Palladio modelling approach, based on [23].

sume that the resulting modeling language is not overly complicated to use and that our approach can also be applied to other design-time modeling languages. iii) Palladio has been proven to support various quality properties including performance [6], reliability [7] or maintainability [24], so it is interesting to see if it can also support analyzing access control. Besides these reasons, we would like to stress that we decouple the analysis from the particular modeling language as already described. Therefore, we do not see our decision as a critical or limiting point.

As can be seen from the illustration of the metamodel in Figure 3, there is only a limited amount of extensions shown by grey elements. These extensions overcome the limitations of the Palladio core language to express behavior in a data-oriented way. The first and most important extension is the introduction of *Data* to the modeling language. To foster integration into the existing modeling language, we do not enforce dedicated data interfaces but reuse the existing operational interfaces consisting of call and return signatures. We assume that data is always exchanged via parameters (which includes return values) in such systems. Therefore, we require that every parameter transports at least one data item. Thus, we can represent that multiple different data items are contained within one parameter without changing the service signature. This enables refining the transported data with low effort. Otherwise, either the service signature would have to be adjusted or a more complex data type modeling would be necessary. Nevertheless, a data item always has an associated data type.

Palladio already provides means for specifying system and user behavior in terms of actions. System behavior is encapsulated in a so-called service effect specification (SEFF). User behavior is encapsulated in a usage description. Roughly said, the actions contained in these behavior descriptions are either

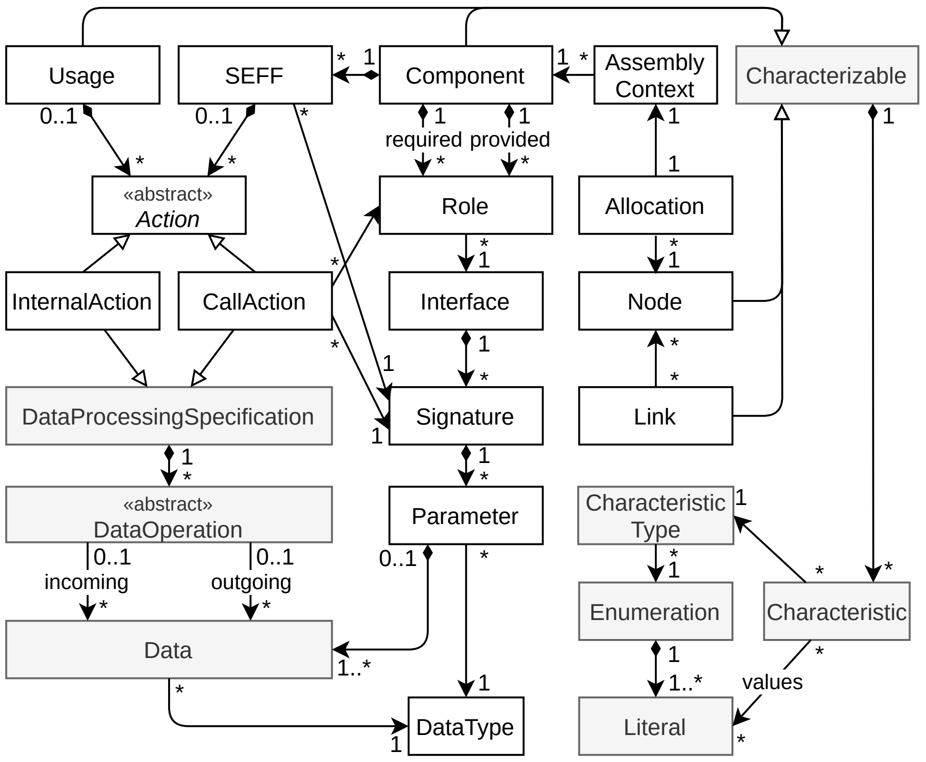


Figure 3: Data-Driven Architecture metamodel given as UML class diagram. Grey elements are extensions to the Palladio core language given by white elements.

i) internal actions taking place within a service or user behavior or are ii) call actions that trigger the behavior of another service. All of these actions can have an effect on the processed data. For instance, an internal action selecting only certain parts of available information might reduce the privacy level from *highly confidential* to *publicly available*. In our running example, the data record of a worker may contain highly sensitive data such as a history of sick days. When selecting only the phone number of the worker, the data might still be sensitive but not as much as the whole data record. To specify this effect on data, we attach a *Data Processing Specification* to the actions. These specifications contain a list of *Data Operations*. These operations consume data, such as the data specified via parameters, and yield data. Other operations can use such yielded data to produce new data and yield this produced data. These operations build a processing chain that eventually yields data that is passed to other services via parameters or back to the caller via return values. We provide a set of predefined data operations that can also be extended by further operations depending on the particular domain.

There are five categories of data processing operations as shown in Table 1. Operations in the source category do not consume data but only yield data. These operations are the start of a data flow. This either covers creating completely new data (*CreateData*) or loading data from data stores (*LoadData*/*LoadAllData*). Obviously, when only yielding data, the sensitivity of data has to be specified explicitly. Operations in the sink category only consume but do not yield data. This covers storing the data (*StoreData*) but also discarding data in various forms (remaining operations). These operations are the end of a data flow. Transmission operations bridge the gap between control flow and data flow by attaching data to parameters and taking data from returns (*PerformDataTransmission*) as well as returning data (*ReturnData*). These operations do not change the sensitivity of data. Relational operations perform operations of relational algebra on data. The effect of operations on data sensitivity depends on the particular data type and has to be specified per data type. Joining (*JoinData*) builds new data from consumed data parts or fragments. Merging data sets (*UnionData*) builds a new data set by merging two data sets. Extracting a data part or fragment (*ProjectData*) builds new data based on incoming data. Selecting data from a data set (*SelectData*) filters data sets based on a selection parameter. The last operation in a dedicated category is *TransformData*. The operation represents a generic data transformation to be specified by the designer. The operation allows the designer to explicitly state the effect of data processing on data properties such as the sensitivity. This is different to previous operations because it does not require general applicable rules to derive the sensitivity anymore but allows to define these rules in particular for one dedicated operation.

We use characteristics to describe properties, such as sensitivity, of data or system parts. As illustrated in Figure 3, a characteristic always has a corresponding type. The type refers to a set of possible values. Because this set is a finite set of discrete values, we use enumerations to describe this value set. With respect to our running example, we specify a characteristic type *privacy level* with the values *public*, *internal use*, *sensitive* and *highly sensitive* with increasing sensitivity. A characteristic selects an arbitrary number of values, which means that these values are available in this particular characteristic. In our running example, it is only useful to have exactly one privacy level, so ev-

Category	Operation	In	Out
Source	CreateData	0	1
	LoadData	0	1
	LoadAllData	0	1
Sink	StoreData	1	0
	DeleteData	1	0
	UserReadData	1	0
	SystemDiscardData	1	0
Transmission	PerformDataTransmission	n	m
	ReturnData	1	0
Relational	JoinData	n	1
	UnionData	n	1
	ProjectData	1	1
	SelectData	1(+n)	1
Characteristics	TransformData	1	1

Table 1: Predefined data processing operations.

ery characteristic only holds one value. Several structural elements within the software architecture are characterizable, which means that they can hold characteristics. In our running example, this is not necessary but in other scenarios, it might be useful to specify roles, clearance levels or criticality of system parts. The characterizable system parts are the following: A usage represents a user, so it is useful to make it characterizable to represent properties of users. A component represents a service provider, for which it is also useful to specify properties. Component instances, i.e., assembly contexts, are allocated to a node, which can also have relevant properties such as a geographical location or an owner. Nodes communicate via links and depending on the scenario it might be necessary to consider the properties of the network connection between nodes.

In our running example, the system service providing the foreman with the phone number of a worker is specified roughly as sketched in Figure 4. Initially, the foreman sends the worker name to a service of his/her *Management Component*. The action-based specification of the service is given in the lower part of the management component. To provide the phone number, the component calls the *DB Component* and extracts it. To specify the data processing more precisely, the designer extends the actions by data specifications. The specification of the call action simply contains a *PerformDataTransmission* operation that receives a list of workers from the database component. The internal action is specified by two actions. The first action selects a particular worker based on the name of the worker from the list of workers. The second action extracts the phone number of that worker. The data specification of the internal action of the database component simply loads all workers from the database *HumanResourceStorage*. All shown operations except for the *Project* operation do not change the privacy level but simply forward it. The *Project* operation effectively lowers the privacy level because it only selects the phone number that is considered less sensitive compared to all information about a worker. The effect of data processing, which is the effect on the privacy level in our running example, is defined for every triple of data processing operation type, input data type and output data type. Because the characteristic types are usually case-specific, the definition of data processing effects are also specified individually per case. However, reusing the effect in cases using the same characteristic types is possible.

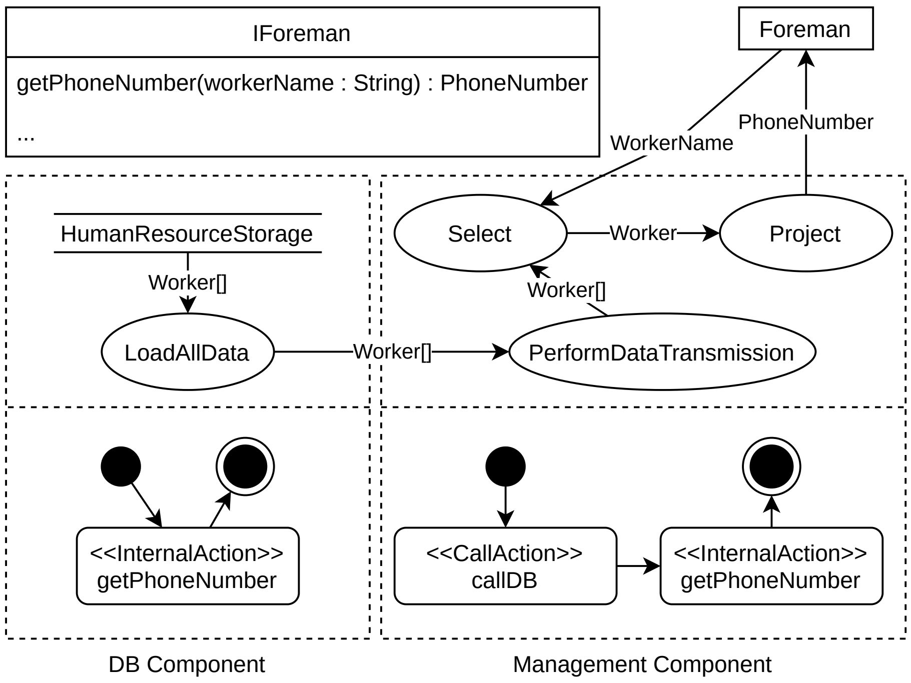


Figure 4: Excerpt of the Palladio model representing the phone number provider used by the foreman.

The data flow analysis traces data including its properties through the system and determines which actor has access to which data. A label propagation algorithm carries out this analysis. The propagated labels are the literals of characteristic types and the propagation rules stem from the data processing effect of the used operators. Afterwards, the raw result of the analysis is transformed into a text file specifying a) the subject that accesses data, b) the type of access, which is always reading in our case, c) the object that is accessed and d) the determined privacy level. The line for the phone number access, for example, looks like this: `foreman;read(phoneNumber);worker;internal-use`. More details on the data flow analysis are given in Section 5.

4 Dynamic Access Control

To specify access control in a highly dynamic environment of Industry 4.0, we base the approach on our previous work [4], where we defined the concept of autonomic *ensembles*. An ensemble is a dynamically formed group of components (they can be both software and hardware components but also, importantly, directly uncontrollable components like people, etc.) that is formed to achieve joint goal or perform a coordinated activity. Components within a single ensemble are selected dynamically at runtime, based on a set of predicates defined in the ensemble.

Using ensembles for specifying access control in a system, an ensemble definition represents a particular situation in a system and defines components taking part in the situation and access rules for these components. The ensembles are established at the moment the particular situation occurs in the system and, in the same way, it ceases its existence when the situation disappears.

Compared to usual component-based approaches, we do not require direct control of the components. Thus, as mentioned above, components can be of any kind—software and hardware but also people or even places.

In both cases of components and ensembles, our approach distinguishes between definitions (types) and instances of components and ensembles. I.e., for components we can have a single type defining a worker or a shift (in the running example) and then multiple instances of them. Similarly for ensembles, we can have an ensemble type defining access to workplace and then as many instances as we have shifts at a particular time.

The ensembles' definitions in our approach represent an access control rule. In more detail, the ensemble definition specifies a situation, identifies components taking roles in the situations and grants/denies access to particular resources.

In the rest of this section, we describe the above shown concepts on the running example and further describe the semantics.

4.1 Specification of the Running Example

For easy creation of access control specifications and rapid testing of our approach, we have created a Scala-based domain specific language (DSL). Listing 1 shows an excerpt of the specification of the running example (described in Section 2). Being defined as an Scala internal DSL, its usage requires at least basic knowledge of the Scala language; on the other hand, the necessary Scala concepts are more-or-less the same as in any other modern programming language.

Scala is an ideal language for rapid creation of DSL, as it has a variable and flexible syntax (compared to more traditional languages like, e.g., Java). Thanks to this feature, it is easy to create new "keywords", which from the implementation perspective are regular methods. When calling a method with a single argument, the round parentheses around arguments can be replaced with curly braces. This is ideal when a particular method takes as its argument another method (or function) and thus its call can be seen as a block of code prefixed by a keyword (in reality, it is a method call with an anonymous function passed as its argument). A particular example in our DSL is, e.g., the "keyword" `situation`, which actually is a call of a method which accepts a boolean function as its argument. These method calls, which are placed directly in the class body, are executed during the class instantiation, i.e., they are part of the default constructor. Another nice feature of Scala is that methods can be used as infix operators. E.g., the `isAfter` operator on line 45 is just a method call on the object `now`.

In our DSL, both the component and ensemble types are modeled as classes while their instances are the particular classes' instances. The particular classes have to extend the Component class or Ensemble class respectively. In the example, there are six components (lines 3–25), which represent the physical components in the running example—namely doors, head-gear dispensers, workers, work places and factories. Each of the components has its attributes (called a *component knowledge* in the terminology of ensembles), however, as we do not directly control the components, we can only observe their values. Here, for example, all of the components have the attribute id and most of them have the position attribute.

The ensembles can be hierarchically nested (an ensemble can contain other ensembles). This means that components, which are members of an ensemble, have to be also members of the parent ensemble. Thus, a top-level ensemble (in our DSL it has to extend the RootEnsemble class) describes a goal of the system as a whole while the sub-ensembles decompose the system into sub-goals, which are easily manageable. A component can be member of many ensembles at the same time (even directly unrelated ensembles) reflecting a common requirement that a single component can be simultaneously in many different situations.

In the example, there are four ensemble types. The top-level ensemble—FactoryTeams (starting at line 27)—represents all the individual ShiftTeam ensembles (line 116). As shifts are specified externally, the FactoryTeams ensemble only models the teams. The FactoryTeams ensemble declares a global constraint for the system that standby workers need to be assigned to the shifts where required and a standby worker cannot be shared among several shifts (lines 117–120). This is necessary, since the selection of standby workers and giving them the respective access control rights is performed by the ensemble system. The FactoryTeam ensemble is instantiated for every factory in the example (line 122).

The ShiftTeam ensemble (lines 28–114) models most of the activities in the system. It is parameterized by the Shift component instance (which defines the shift). The individual access control rules are represented by sub-ensembles. The ensemble declares 5 lists (lines 29–32) which aid identification of workers during their selection by sub-ensembles. In general, the ShiftTeam's sub-ensembles can be divided into ensembles, which (i) assign permission to individual workers and which (ii) notify workers about selection for or removal from a shift. The former ones are the ensembles AccessToFactory, AccessToDispenser, AccessToWorkplace, while the latter ones are CancellationOfWorkersThatAreLate and AssignmentOfStandbys. The NotificationAboutWorkersThatArePotentiallyLate ensemble performs both functions. All of them have the same structure, which is as follows. First, there is a definition of the *situation*, which defines a spatial and temporal condition under which the ensemble is formed. For the AccessToFactory, the current time has to be in the interval of start of the shift minus 30 minutes and end of the shift plus 30 minutes. It is similar for AccessToDispenser, but there is a different time interval. In the case of the AccessToWorkplace, there is an extra condition (in addition to the situation) that the workers must have a headgear from the dispenser expressed as a selection of the shift workers with the headgear (line 52). All these three ensembles assign (lines 39, 48, and 57) the particular permissions (to enter the factory, use the dispenser, enter the workplace) to the workers selected by the conditions.

The NotificationAboutWorkersThatArePotentiallyLate ensemble detects workers assigned to the shift but not present in the factory (line 61) 20 minutes before start of the shift (line 63). For these workers, the ensemble notifies the particular foreman that they are late and allows the foreman to see the workers' phone numbers (the foreman can call them to "hurry up"—line 65) and their distance[1] from the factory (to see whether there is a chance to come in time yet—line 66).

The CancellationOfWorkersThatAreLate ensemble is similar to the previous one, but it detects workers, which are late even 15 minutes before start of the shift (line 73), and notifies them that they are canceled from the shift (line 75).

Finally, the AssignmentOfStandbys ensemble selects and notifies the standby workers that replace the canceled late workers. To do so, there is defined the sub-ensemble StandbyAssignment (line 79), which selects a suitable standby worker for a particular canceled worker (the selected standby has to have the same capabilities as the canceled one). The sub-ensemble is instantiated for each canceled worker (lines 85 and 86). The constraint (line 91) requires that a single standby worker is not used as a replacement for several workers. Within the given time interval (the situation definition at line 88), the ensemble notifies the selected workers to come (line 93) and the foreman (line 94) of the particular shift.

The ensemble NoAccessToPersonalDataExceptForLateWorkers expresses access control assertions (i.e., forbidden situations). Such assertions serve as safe-guards to detect potential inconsistencies in the specification. At runtime, they are used to verify that access control rules determined by ensembles. In detail, it is described and discussed in the following section.

From a technical point of view, a part of the ensembles in the example is declared as classes while other ensembles are declared as objects. This is an exploitation of another feature of the Scala language—the object is a class definition with a singleton instance. The name after the object keyword refers to the instance. The ensembles that are needed in a single instance only are therefore defined as objects.

To summarize our DSL, there are two predefined classes (Ensemble and Component) for extensions. Plus, there are six new "keywords" (actually methods) that are situation, constraints, rules, allow, deny, and notify. The semantics of these keywords will be explained in the following section.

```
1  class TestScenario(scenarioParams: TestScenarioSpec) extends Model with ModelGenerator {
2    ...
3    class Door(val id: String, val position: Position) extends Component
4    class Dispenser(val id: String, val position: Position) extends Component
5    class Worker(
6      val id: String, var position: Position,
7      val capabilities: Set[String], var hasHeadGear: Boolean
8    ) extends Component {
9      def isAt(room: Room) = room.positions.contains(position)
10   }
11   class WorkPlace(
12     id: String, positions: List[Position], entryDoor: Door
13   ) extends Room(id, positions, entryDoor) {
14     var factory: Factory = _
15   }
16   class Factory(
17     id: String, positions: List[Position], entryDoor: Door,
18     val dispenser: Dispenser, val workPlaces: List[WorkPlace]
19   ) extends Room(id, positions, entryDoor)
20   class Shift(
21     val id: String, val startTime: LocalDateTime,
```

[1] As the exact position of a worker outside the factory can be potentially very sensitive information, in our implementation we are using abstracted values like *close*, *far*, etc.

```
  val endTime: LocalDateTime, val workPlace: WorkPlace,
  val foreman: Worker, val workers: List[Worker],
  val standbys: List[Worker], val assignments: Map[Worker, String]
) extends Component

class FactoryTeam(factory: Factory) extends RootEnsemble {
  class ShiftTeam(shift: Shift) extends Ensemble {
    val canceledWorkers = shift.workers.filter(wrk => wrk notified
          AssignmentCanceledNotification(shift))
    val calledInStandbys = shift.standbys.filter(wrk => wrk notified CallStandbyNotification(shift))
    val availableStandbys = shift.standbys diff calledInStandbys
    val assignedWorkers = (shift.workers union calledInStandbys) diff canceledWorkers

    object AccessToFactory extends Ensemble {
      situation {
        (now isAfter (shift.startTime minusMinutes 30)) &&
          (now isBefore (shift.endTime plusMinutes 30))
      }
      allow(shift.foreman, "enter", shift.workPlace.factory)
      allow(assignedWorkers, "enter", shift.workPlace.factory)
    }

    object AccessToDispenser extends Ensemble {
      situation {
        (now isAfter (shift.startTime minusMinutes 15)) &&
          (now isBefore shift.endTime)
      }
      allow(assignedWorkers, "use", shift.workPlace.factory.dispenser)
    }

    object AccessToWorkplace extends Ensemble {
      val workersWithHeadGear = (shift.foreman :: assignedWorkers).filter(wrk => wrk.hasHeadGear)
      situation {
        (now isAfter (shift.startTime minusMinutes 30)) &&
          (now isBefore (shift.endTime plusMinutes 30))
      }
      allow(workersWithHeadGear, "enter", shift.workPlace)
    }

    object NotificationAboutWorkersThatArePotentiallyLate extends Ensemble {
      val workersThatAreLate = assignedWorkers.filter(wrk => !(wrk isAt shift.workPlace.factory))
      situation {
        now isAfter (shift.startTime minusMinutes 20)
      }
      workersThatAreLate.foreach(wrk => notify(shift.foreman,
            WorkerPotentiallyLateNotification(shift, wrk)))
      allow(shift.foreman, "read.personalData.phoneNo", workersThatAreLate)
      allow(shift.foreman, "read.distanceToWorkPlace", workersThatAreLate)
    }

    object CancellationOfWorkersThatAreLate extends Ensemble {
      val workersThatAreLate = assignedWorkers.filter(wrk => !(wrk isAt shift.workPlace.factory))
      situation {
        now isAfter (shift.startTime minusMinutes 15)
      }
      notify(workersThatAreLate, AssignmentCanceledNotification(shift))
    }

    object AssignmentOfStandbys extends Ensemble {
      class StandbyAssignment(canceledWorker: Worker) extends Ensemble {
        val standby = oneOf(availableStandbys)
        constraints {
          standby.all(_.capabilities contains shift.assignments(canceledWorker))
        }
      }
      val standbyAssignments = rules(canceledWorkersWithoutStandby.map(wrk => new
            StandbyAssignment(wrk)))
      val selectedStandbys = unionOf(standbyAssignments.map(_.standby))
      situation {
        (now isAfter (shift.startTime minusMinutes 15)) && (now isBefore shift.endTime)
      }
      constraints {
        standbyAssignments.map(_.standby).allDisjoint
      }
```

```
        notify(selectedStandbys.selectedMembers, StandbyNotification(shift))
        canceledWorkersWithoutStandby.foreach(wrk => notify(shift.foreman,
              WorkerReplacedNotification(shift, wrk)))
      }

      object NoAccessToPersonalDataExceptForLateWorkers extends Ensemble {
          val workersPotentiallyLate =
              if ((now isAfter (shift.startTime minusMinutes 20)) && (now isBefore shift.startTime))
                  assignedWorkers.filter(wrk => !(wrk isAt shift.workPlace.factory))
              else Nil
          val workers = shift.workers diff workersPotentiallyLate
          deny(shift.foreman, "read.personalData", workers, PrivacyLevel.ANY)
          deny(shift.foreman, "read.personalData", workersPotentiallyLate, PrivacyLevel.SENSITIVE)
      }
      rules(
        // Grants
        AccessToFactory, AccessToDispenser, AccessToWorkplace,
        NotificationAboutWorkersThatArePotentiallyLate,
        CancellationOfWorkersThatAreLate, AssignmentOfStandbys,
        // Assertions
        NoAccessToPersonalDataExceptForLateWorkers
      )
    }

    val shiftTeams = rules(shiftsMap.values.filter(shift => shift.workPlace.factory ==
          factory).map(shift => new ShiftTeam(shift)))
    constraints {
      shiftTeams.map(shift => shift.AssignmentOfStandbys
         .selectedStandbys).allDisjoint
    }
  }
  val factoryTeams = factoriesMap.values.map(factory => root(new FactoryTeam(factory)))
}
```

Listing 1: Access control specification

4.2 Semantics

Formally, the specification describes a constraint optimization problem. A solution to the problem determines which ensemble instances are to be formed and which are their members. Every ensemble can be instantiated multiple times (once for every instance of the situation it reflects). To connect the instance of an ensemble with a particular instance of the situation, ensembles are typically parameterized (using constructor arguments). Only in case an ensemble is a singleton, the parameters are missing (e.g., in case of CancellationOfWorkersThatAreLate). An ensemble instance is created only if the situation it reflects happens. Formally, it is created if the condition specified in the situation block is true.

Each ensemble identifies its members components either directly (e.g., in the AccessToFactory and AccessToDispenser ensembles) or by listing potential members and formulating constraints governing their selection (e.g., in the AssignmentOfStandbys and StandbyAssignment). The identification of potential members is done through functions oneOf and unionOf. The result is a set variable that represents a selection from the set of components given to these functions as a parameter. The valuation of these variables is constrained by the constrains blocks, which make it possible to express the constraints using common logical and set operators.

When ensembles are nested, the parent ensemble identifies all potential sub-ensembles. This is done with the rules statement. The sub-ensembles are however only instantiated if their situation condition holds. Ensembles are properly nested, thus a sub-ensemble instance can be only instantiated if the parent ensemble is instantiated, too. Formally, the rules statement creates a condition whose existence of a sub-ensemble instance implies the existence of the containing ensemble.

Nested ensembles form a tree structure. The root is identified using the root statement.

If an ensemble is instantiated, its allow statements determine the permissions. Each allow permission is formed as a triple <subject,verb,object>. Our access control model denies every access request, unless it is explicitly allowed. Thus, deny rules by themselves are not needed. Nevertheless, we use them in our approach as runtime assertions. We say that a specification is consistent if there are no conflicting allow and deny rules. Each deny permission is formed as a triple <subject,verb,object> or as a quadruple <subject,verb,object,privacy_level>. The form with the privacy relates to what we presented in Section 3. It makes it possible to restrict the deny rule only to data at a particular level of sensitivity. This is useful to protect data privacy by means of access control because it allows us to dynamically assign and restrict access to data based on changing situations in the environment.

An instantiated ensemble may also perform notifications (specified using notify) statement. This is needed to let a user know about permissions that have been dynamically assigned and revoked. The notify statement has the form <target,message>. The semantics is that every such pair is notified only once. All subsequent notifications for the same pair are ignored.

5 Application Scenarios

We have already presented the models used to describe the static and dynamic part of our approach in the previous sections, but have only roughly described how these models were used to make access control decisions. Therefore, we briefly give an overview on how both parts of the approach are combined to decide about access to information or locations in Section 5.1. Afterwards, we give insights into the corresponding tool support that we used to realize our running example. We will start with the tooling for static analyses in Section 5.2 and continue with tooling for dynamic access control enforcement in Section 5.3.

5.1 Overview of the Combined Approach

The overall goal of the combined approach is to decide about virtual or physical access requests during runtime. To do so, there is a decision point that receives access requests from various services or locations that the decision point answers. This decision point is the last action in the overview given in Figure 5. To properly decide about requests, the decision point needs the set of applicable rules, i.e., the policy that specifies allowed and denied access requests.

Which rules are applicable depends on the current system state and system context. The ensembles of the runtime approach consider these dynamically changing situations to filter all defined rules for the applicable ones in a solving process. The solving process needs the ensembles, the system or context state and the privacy levels of processed data. The state information originates from a set of various probes that collect state information. The ensembles contain

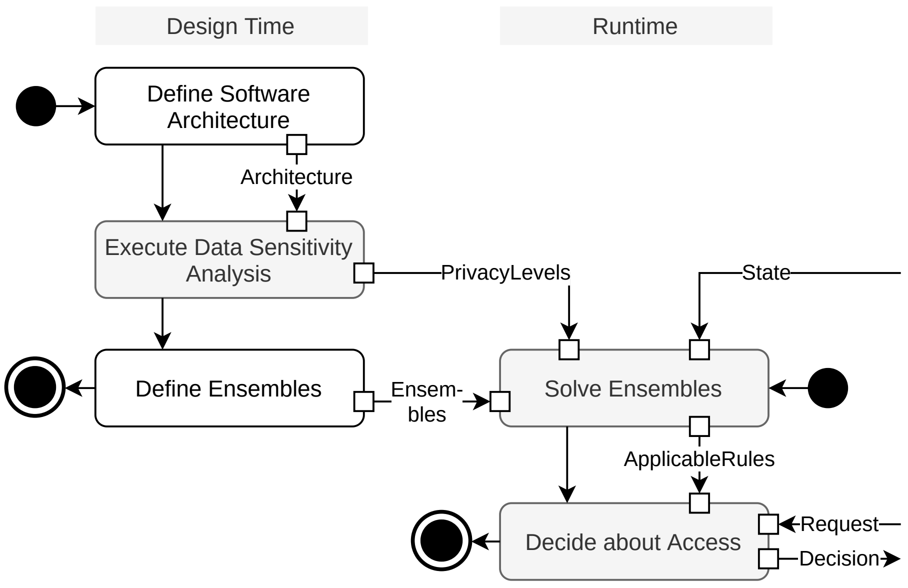


Figure 5: Overview on combination of static design time and dynamic runtime parts of approach given as UML activity diagram. White elements are executed manually, grey elements are executed automatically.

the access rules and criteria for when the rules shall be applied during design time. The privacy levels are the result of a sensitivity analysis executed on the software architecture during design time. The privacy levels serve as additional source of information for deciding on the applicability of a rule as we will show later.

As can be seen in Figure 5, the only manual steps in the process sketched before are the definition of the software architecture and the ensembles. Both steps have to be done during design time. During runtime, the results of these manual steps are used.

There are two roles involved in defining both artifacts: A software architect is responsible for defining the system architecture including the data processing and turning access control policies into ensembles (in cooperation with an access control expert). The software architect has to know the architectural description language Palladio and its extension for data processing as well as the internal DSL for specification of ensembles. An access control expert supports the software architect by defining the access control policies as well as additional data processing operations and data properties. Access control experts also have the possibility to extend or to define new design-time analyses. In our scenario, the sensitivity analysis is predefined, so the access control expert does not have to define that particular analysis anymore. The analysis of the static design-time analysis is specified in a logic programming language that the access control expert has to know. The analysis definition uses the processing operations and data properties to determine the privacy levels. In the running example, the privacy levels are the data properties. The combination of data is an example of a data processing operation. The effect of such a combination is that the highest privacy level available on inputs is applied to the output. It is reasonable to have a dedicated access control expert defining these artifacts because a software architect might not have the required expertise to do so. The software architect can assume that the access control expert covers all important properties and processing operations that the architect can reuse afterwards.

5.2 Palladio Design-Time Tooling

The overall process to determine data sensitivity during design time is shown in Figure 6. We start with a conventional system design created using the non-extended version of Palladio. After modeling the architecture, we expect the architect to extend it with data processing specifications that include specifying data, data exchange between services and data processing steps by using the modeling language presented in Section 3. The modeling language is available as metamodel[2] and can be used by a prototypical graphical editor[3]. To decouple our analysis approach from the particular modeling language, we defined a dedicated analysis model[4] that is tailored to our analysis and only includes the essential aspects of the design. We later describe this model and how it uses the effects of data processing for every operation that we discussed in Section 3. The mapping of the architecture to the analysis model is done automatically by a model-to-model transformation[5]. The actual analysis is carried out by a logic program. Again, the transformation into the logic program is done automatically by a model-to-text transformation[6]. Architects now can execute the data sensitivity analysis within the logic program. Internally, an interpreter takes the logic program and executes a query for the sensitivity. In the last step, the sensitivities are extracted in form of privacy levels from the analysis result. This requires mapping system elements from the logic program back to the architecture. This is done automatically by looking up the elements in the traces of the model transformations. Eventually, the privacy levels are written into a file to be used during runtime by the dynamic access control analysis.

The analysis metamodel as shown in Figure 7 is a description of the system design tailored to data flow analyses. A system consists of system usages and operations. System usages are calls to the system by external actors. Operations are processing steps of the system that consume and produce data. The interface of such an operation is given by variables. There are variables representing parameters, returns and states. Operation calls from external actors to operations, as well as calls between operations transport data by defining assignments to target states or parameters. The operations specify their effect by assignments to returns. Assignments can assign constant values but can also use parameters or states to derive a value. A value represents one particular characteristic such as the privacy level *sensitive*. In terms of label propagation, these values are labels and the assignments are the propagation functions. For a sake of simplicity, the elements representing the value types and the terms for the assignments are not shown in Figure 7.

[2]`https://github.com/Trust40-Project/Palladio-Addons-DataProcessing-MetaModel`

[3]`https://github.com/Trust40-Project/Palladio-Addons-DataProcessing-Editor`

[4]`https://github.com/Trust40-Project/Palladio-Addons-DataProcessing-PrologModel`

[5]`https://github.com/Trust40-Project/Palladio-Addons-DataProcessing-AnalysisTransformation`

[6]`https://github.com/Trust40-Project/Palladio-Addons-DataProcessing-PrologModel/tree/master/bundles/org.palladiosimulator.pcm.dataprocessing.prolog.transformation`

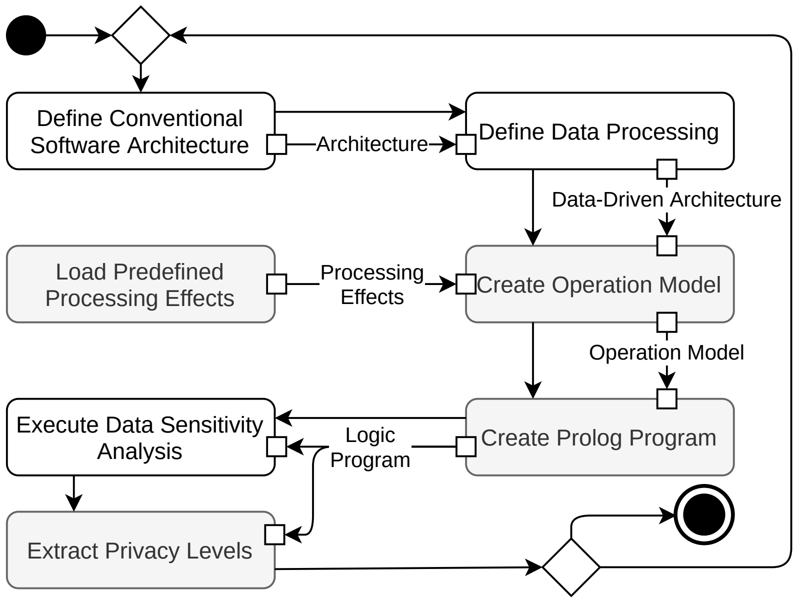


Figure 6: Process for deriving privacy levels during design time given as UML activity diagram. White elements are executed manually, grey elements are executed automatically.

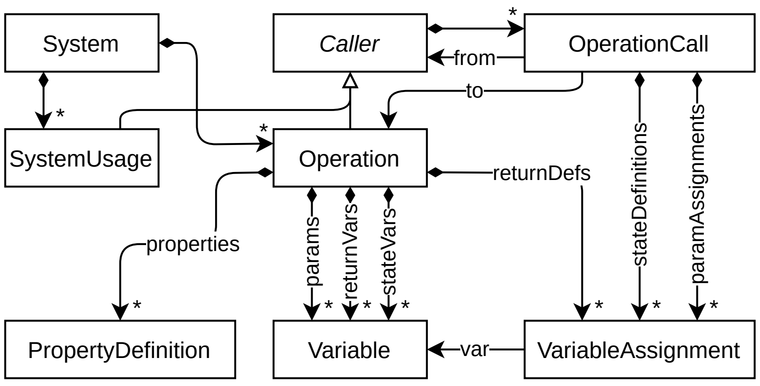


Figure 7: Analysis metamodel tailored to data flow analyses.

The automated mapping from the software architecture to the analysis model is as follows. There is always exactly one system for each modeled software architecture. Every usage, i.e., every user, is mapped to a system usage. Every data processing operation is mapped to an operation. All data assigned to parameters in the architecture become parameter variables of the corresponding operation. All data assigned to returns in the architecture become return variables of the corresponding operation. The variable assignments of return definitions of an operation are given by the processing effects of the data processing operations in the software architecture. For instance, the projection operation separating the phone number from a worker data record has the effect of reducing the sensitivity. The variable assignments of parameter definitions are always copy assignments that just pass variables as they are to the next operation. The next operation is determined by the data dependencies in the software architecture. If a data processing operation requires a certain data item that another operation emits, there is a data dependency. This data dependency is mapped to an operation call from the emitting operation to the receiving operation. There are variables and property definitions for every possible value of all possible characteristic types. For instance, there is a variable for every privacy level for the return value of the phone number getter. An assignment of a truth value to such a variable indicates whether the privacy level is available.

An automated model-to-text transformation carries out the mapping to the logic program. Most parts of the transformation simply map elements one by one to logical facts. The remainder of the transformations adds logical rules to perform the label propagation. Because both steps are mostly straight forward, we omit a detailed discussion of the transformation to save space. Instead, we focus on the query for the generated logic program and explain relevant parts of the logic program while explaining the query. The goal of the query shown in Listing 2 is to find the privacy level `VAL` of an output `VAR` of an operation `OP` with data type `T`. Please note that the resolution algorithm of Prolog is capable of finding all possible solutions for a query, which means that all possible bindings for the given variables are found. This is exactly what we are aiming for because we want to get all privacy levels for all outputs of all operations in the system. To do so, we define that the interesting characteristic type is the privacy level in line 1. In line 2, we find a return variable `VAR` with the data type `T` for the operation `OP`. Again, this line automatically considers all returns of all operations. After that line, all of these variables are bound, i.e. have a value. Lines 3-4 ensure that we have a valid call stack `S`, which is just a list of called operations and operation calls. The call stack has to have the current operation `OP` on the top, which means that the previous call has been made to this operation and there is no call after that call. In line 5, we determine the privacy level `VAL` for the return variable `VAR` for the given call stack `S`. The call stack is important here because operations can be called from various places, which can have an effect on the privacy level. For instance, consider a simple echo operation that just returns what it received. If the operation is called with highly sensitive data, it will return highly sensitive data. Therefore, it is crucial to know the call stack for determining the privacy level. Under the hood, the `returnValue` rule goes back the call stack until it can find a privacy level by applying the label propagations we mentioned before as part of the analysis metamodel. One out of many results is given in lines 7 to 13. It just reports all bindings to all variables mentioned in the query. In line 12, we can see that the privacy level

Listing 2: Queries to logic program for determining privacy levels of yielded data.

```
?- ATTR = 'PrivacyLevel',
operationReturnValueType(OP, VAR, T),
S = [OP|_],
stackValid(S),
returnValue(S, VAR, ATTR, VAL).

ATTR = 'PrivacyLevel',
OP = 'getPhoneNumber',
VAR = 'RETURN',
T = 'PhoneNumber',
S = ['getPhoneNumber'|...],
VAL = 'internal-use';
...
```

of the phone number is now *internal-use*.

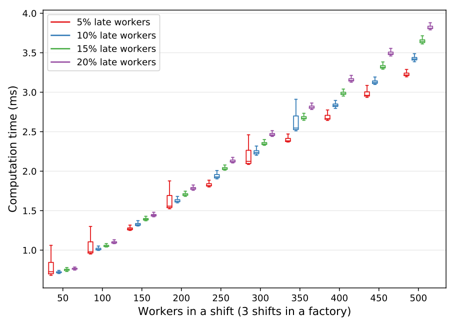
For the running example, we can reuse the shown query, so no manual definitions are necessary. However, access control experts can define further queries or extend the existing query by using a set of predefined logic predicates. This certainly requires some effort in learning the predicates and also requires some limited knowledge in logic programming. However, supporting customize queries provides huge flexibility and allows using the analysis approach for other purposes and scenarios. As we have shown in another publication [28], such flexible query definitions can not only be used to collect information such as privacy levels but can also look for design issues violating confidentiality requirements in a software architecture. Alternatively, a domain specific language (DSL) can be used to define data flow constraints which requires no knowledge about the analysis process [15].

In the last step, we create a CSV file based on the returned results. The CSV file consists of the columns subject, action, object and privacy level. In the chosen example, the subject is the foreman. The action is reading the phone number. The object is a worker. The privacy level is *internal-use*. The created CSV file is passed to the tooling considered with dynamic access control during runtime that we describe in the following.

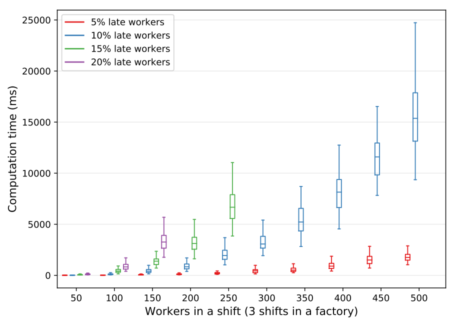
5.3 Runtime Decision Making

For runtime decision making (also described in [3]), we base our implementation on the standard MAPE-K loop [20]. The loop phases work as follows. (a) Monitoring: data about the current situation are collected (in the running example, a position of all the workers, data about the shifts, etc. (b) Analysis: Ensembles are instantiated according to the observed situation. Then the specification of ensembles is translated into a constraint satisfaction problem (CSP). A CSP solver is then applied to find a model for the logical theory described by the CSP and thus the ensemble instances are determined. (c) Planning: Determined ensemble instances provides particular access grants. (d) Execution: The access grants are applied to the system and thus system is updated to conform with the current situation observed in phase (a).

Given the fact that we employ a CSP solver to determine the ensemble



(a) 17 minutes to the shift



(b) 13 minutes to the shift

Figure 8: Evaluation results.

instances and their member components, a question naturally arises about the scalability of such an approach as the CSP solving has inherently exponential complexity. To showcase how our approach scales, we have conducted a series of experiments of different scenario sizes and two points of time in the scenario: (a) 17 minutes before the shift and (a) 13 minutes before the shift.

The different scenario sizes are achieved by varying the number of workers in a shift (from 50 to 500) and the percentage of late workers that are canceled from the shift (from 5% to 20%). There are 3 shifts running at the same time. The shifts compete for the same standby workers. The number of standby workers available is equals to $no\_of\_workers\_in\_shift * percentage\_of\_late\_workers * 5$.

These two times were selected because they mark two major cases in the scenario. Case (a) corresponds to the time when multiple ensembles are instantiated, but all of these perform selection of member components directly. Thus no complex constraint optimization is needed. Case (b), on the other hand, involves the assignment of standbys—i.e., one unique suitable standby for each worker that is late. This assignment represents an optimization problem which is inherently NP hard—it is essentially a scheduling problem.

From the implementation perspective, we deal with both these two cases the same way—we translate the specification to a constraint solving problem (CSP) and use a CSP solver to figure out, which ensemble instances should exist and which components belong to which ensemble instance. Having this in place, the ensemble instances determine the allow, deny and notify rules. In case of the direct selection in case (a), the constraint problem contains essentially no alternatives to select from, thus, the CSP solving amount to traversing the constraint graph and grounding the variables to their only permissible value. Thus, even thought it is processed by the CSP solver, the computation time is essentially linear to the number of components and potential ensemble instances. In case (b), there are multiple mutually exclusive options that the CPS solver has to traverse. This is, as expected, exponential, but only in the dimension that determines variants—in our case the number of workers that have been cancelled (and thus with the number of standby workers shared between the three shifts).

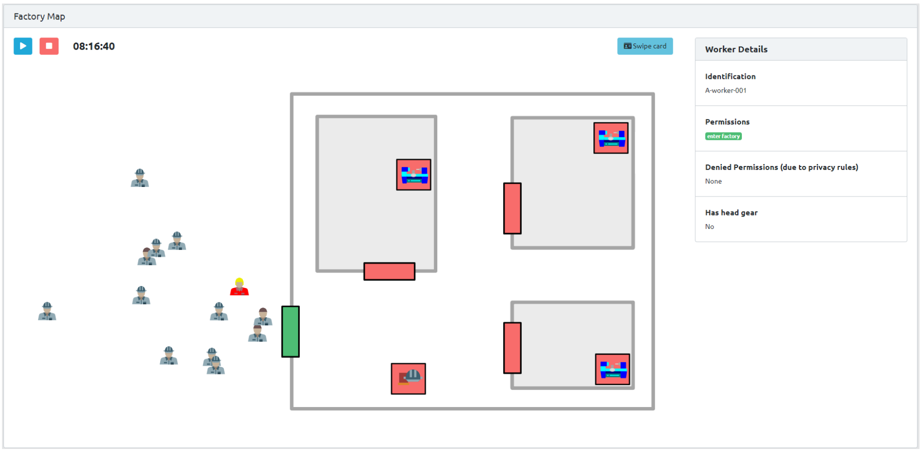
We conducted the scalability experiments on Intel(R) Xeon(R) CPU E5-2660, running on 2.20GHz. In the case of #1, we performed a warmup of 1000 computations and collected 10000 measurements for each size of the scenario. In the case of #2, we performed a warmup of 10 computations and collected 100 measurements for each scenario size. We excluded computations which exceeded 60 seconds. The results are shown in Fig. 8a and Fig. 8b.

In case (a), the time needed to resolve the ensemble instances scales linearly with the number of workers in a shift. Given the fact that 3 shifts are evaluated together, we can easily compute the access rules for 1500 workers in approx. 3.5 milliseconds. Case (b) exhibits exponential growth in time with the increasing percentage of late workers. Nevertheless, even for 10% of late workers, it is possible to assign access grants to 1500 persons in 15 seconds.

The same assignment process is performed for each factory in our running example. As each factory has its own pool of shared standby workers, the ensembles (and consequently the access rules) can be evaluated independently (as it is so in our test case) and in parallel. Thus, the number of factories does not have a significant impact on the computation time, which makes it possible to scale our use-case to arbitrarily large instances—assuming that the size of a shift remains below 500 workers and the percentage of workers that do not come to the shift is below 10%, which we believe is a realistic assumption.

5.3.1 Visualization

To allow for rapid development of access control specifications and immediate observations of results, we have developed a visualization, which provides not only a live view of the simulations, but can also be used for visualization of actual real-life data. Figure 9 shows a screenshot from the visualization of the running example where most of the workers in the factory were simulated and the highlighted (in red) worker showed an actual person requiring interaction with actual devices (access card readers, etc.).



Worker Details

Identification
A-worker-001

Permissions
enter factory

Denied Permissions (due to privacy rules)
None

Has head gear
No

Figure 9: Demo visualization.

The visualization has been developed using our tool IVIS [8], which provides a framework for easy creation of IoT related visualization.

6 Related Work

In this section we discuss work in several areas of research related to our approach.

Access Control Types: Access control is one way to protect the resources of the system. It regulates who can access protected resources and therefore increase the confidentiality of the system. One established way of access control is Role-based Access Control RBAC [13]. It groups users to roles and abstracts from the underlying user, which increases the comprehensibility of access policies. However, the sole focus on roles does not allow to easily model fine-grained access control policies that depend on the environment, like in our running example, the period for entering the factory. Organisation-based Access Control (OrBAC) [18] considers more contextual information, by explicitly modelling context information [9]. Originally, it could not support multiple organizations. However, newer approaches [35, 32] exist for handling multiple organizations, as they might exist in Industry 4.0. In contrast to our approach, their analysis does not support data flow definitions or analysis based on the software architecture. Another approach considering the context for access control is Attribute-based Access Control (ABAC) [16]. Here different Boolean values can form logical conjunctions to model access policies.

Self-adaptive Access Control Policies: Self-adaptive RBAC (saRBAC) [30] models a system by using Markov chains to determine unusual user behavior. On discovery, it automatically adapts the access policies to mitigate potential attacks. In contrast to our approach, the system's architecture is not considered, and only the user behavior is relevant for the adaption. Verma et al. [34] describe an approach, which provides a policy generation for dynamically established coalitions. This is similar to our dynamic approach. However, the coalitions consist only of people with the same shared goal. Additionally, they are not dynamically described as in our approach. Bailey et al. [5] introduce an approach for the management of dynamic policy adaption and optimization using a MAPE-K loop. While this is similar to our dynamic analysis, we use a unified modelling approach for components under direct control and those beyond direct control, such as humans.

Model-driven Approaches for Confidentiality: Nguyen et al. [22] provide an overview of different model-driven security analysis approaches. They conclude that most model-driven approaches analyze confidentiality or access control, similar to our approach. The design-time sensitivity analysis basically is a confidentiality analyses that contributes to the overall access control analysis provided by the complete approach. In the following, we only discuss closely related approaches as examples. UMLSec [17] is an UML profile extensions for annotating security properties. These properties then can be analyzed with the CARiSMA [1] tool. In contrast to our approach, they work on the control flow and have no coupling to dynamic runtime policy management. Another UML extension is SecureUML [21]. It uses internally an RBAC approach, which can be extended with OCL statements to support dynamic access control. It also provides an export function to use the policies during runtime. However, they provide no analysis based on the data flow. Also, since their export functions generate specific Java code, it cannot be reused easily in non-Java environments. The iFlow [19] approach is an information flow analysis, which also uses UML profiles. Another confidentiality analysis is SecDFD [33]. Similar to our static analysis, they analyze the system based on the data flow. However, in contrast to our approach, there is no coupling to a dynamic runtime analysis. R-PRIS [26] investigates changes during runtime, which might introduce confidentiality issues. It uses a runtime model, which is compared to a set of privacy rules. However, R-PRIS [26] only considers the location to determine the confidentiality.

Code-based data flow analysis: Similar to our static analysis, which analyzes the data flow based on the architectural model, there are approaches that analyze them on source code. Joana [31] and KeY [2] are two typically representatives for this. While they do not need architectural models for the analysis, they need the full source code and are specific to one language. Especially in Industry 4.0 environments, where multiple different software systems interact, it might be complicated to apply them.

7 Conclusion and Outlook

We presented an approach to realize dynamic access control policies with a focus on Industry 4.0 systems that covers virtual and physical access control. The combination of both access control types has the potential to strengthen the overall protection of data and physical entities. The approach consists of two parts. First, there is a static design-time analysis to determine the sensitivity of exchanged data in the system. Second, a runtime policy decision point continuously evaluates the dynamic access control policies using the system state, information about actors as well as precalculated data sensitivity. We demonstrated that our approach was capable of making appropriate access control decision in presence of a dynamically changing execution context by applying the approach to a realistic scenario in Industry 4.0.

In the future, we will consider not only dynamic changes in the system or environment but also dynamic changes in the policies required to handle new situations. Adjusting the policies might be necessary because a new situation arises and there needs to be a policy change in order to keep the system and its processes at least partially running. This requires detecting upcoming situations and reacting to them. We will address this challenge as part of the FluidTrust[7] project.

Acknowledgement

This work was funded by the DFG (German Research Foundation) – project number 432576552, HE8596/1-1 (FluidTrust). It was also supported by funding from the topic Engineering Secure Systems of the Helmholtz Association (HGF) and by KASTEL Security Research Labs (46.23.03). The work was also partially supported by the Czech Science Foundation project 20-24814J and partially supported by Charles University institutional funding SVV 260451.

[7]`https://fluidtrust.ipd.kit.edu`

References

[1] Amir Shayan Ahmadian et al. "Model-Based Privacy Analysis in Industrial Ecosystems". In: *Modelling Foundations and Applications - 13th European Conference, ECMFA@STAF 2017, Marburg, Germany, July 19-20, 2017, Proceedings.* Ed. by Anthony Anjorin and Huáscar Espinoza. Vol. 10376. Lecture Notes in Computer Science. Springer, 2017, pp. 215–231. DOI: 10.1007/978-3-319-61482-3_13.

[2] Wolfgang Ahrendt et al., eds. *Deductive Software Verification - The KeY Book - From Theory to Practice.* Vol. 10001. Lecture Notes in Computer Science. Springer, 2016. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6.

[3] Rima Al Ali et al. "Dynamic security rules for legacy systems". In: *Proceedings of the 13th European Conference on Software Architecture, ECSA 2019, Paris, France, September 9-13, 2019, Companion Proceedings (Proceedings Volume 2),* ed. by Laurence Duchien et al. ACM, 2019, pp. 277–284. DOI: 10.1145/3344948.3344974.

[4] Rima Al Ali et al. "Toward autonomically composable and context-dependent access control specification through ensembles". In: *Int. J. Softw. Tools Technol. Transf.* 22.4 (2020), pp. 511–522. DOI: 10.1007/s10009-020-00556-1.

[5] Christopher Bailey, David W. Chadwick, and Rogério de Lemos. "Self-adaptive federated authorization infrastructures". In: *J. Comput. Syst. Sci.* 80.5 (2014), pp. 935–952. DOI: 10.1016/j.jcss.2014.02.003.

[6] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. "The Palladio component model for model-driven performance prediction". In: *J. Syst. Softw.* 82.1 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066.

[7] Franz Brosch et al. "Architecture-Based Reliability Prediction with the Palladio Component Model". In: *IEEE Trans. Software Eng.* 38.6 (2012), pp. 1319–1339. DOI: 10.1109/TSE.2011.94.

[8] Lubomír Bulej et al. "IVIS: Highly customizable framework for visualization and processing of IoT data". In: *46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, Portoroz, Slovenia, August 26-28, 2020.* IEEE, 2020, pp. 585–588. DOI: 10.1109/SEAA51224.2020.00095.

[9] Frédéric Cuppens and Alexandre Miège. "Modelling Contexts in the Or-BAC Model". In: *19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA.* IEEE Computer Society, 2003, pp. 416–425. DOI: 10.1109/CSAC.2003.1254346.

[10] Mohammad Dastbaz. "Industry 4.0 (i4.0): The Hype, the Reality, and the Challenges Ahead". In: *Industry 4.0 and Engineering for a Sustainable Future.* Ed. by Mohammad Dastbaz and Peter Cochrane. Cham: Springer International Publishing, 2019, pp. 1–11. ISBN: 978-3-030-12953-8. DOI: 10.1007/978-3-030-12953-8_1.

[11] Jyoti Deogirikar and Amarsinh Vidhate. “Security attacks in IoT: A survey”. In: *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*. 2017, pp. 32–37. DOI: 10.1109/I-SMAC.2017.8058363.

[12] David F. Ferraiolo et al. *Policy Machine: Features, Architecture, and Specification*. en. Tech. rep. NIST IR 7987r1. National Institute of Standards and Technology, Oct. 2015, NIST IR 7987r1. DOI: 10.6028/NIST.IR.7987r1.

[13] David F. Ferraiolo et al. “Proposed NIST standard for role-based access control”. In: *ACM Trans. Inf. Syst. Secur.* 4.3 (2001), pp. 224–274. DOI: 10.1145/501978.501980.

[14] Steven Furnell, ed. *Securing information and communications systems: principles, technologies, and applications*. en. Artech House computer security series. Boston: Artech House, 2008. ISBN: 978-1-59693-228-9.

[15] Sebastian Hahner et al. “Modeling Data Flow Constraints for Design-Time Confidentiality Analyses”. In: *18th IEEE International Conference on Software Architecture Companion, ICSA Companion 2021, Stuttgart, Germany, March 22-26, 2021*. IEEE, 2021, pp. 15–21. DOI: 10.1109/ICSA-C52384.2021.00009.

[16] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. “Attribute-Based Access Control”. In: *Computer* 48.2 (2015), pp. 85–88. DOI: 10.1109/MC.2015.33.

[17] Jan Jürjens. *Secure systems development with UML*. Springer, 2005. ISBN: 978-3-540-00701-2. DOI: 10.1007/b137706.

[18] Anas Abou El Kalam et al. “Organization based access contro”. In: *4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), 4-6 June 2003, Lake Como, Italy*. IEEE Computer Society, 2003, p. 120. DOI: 10.1109/POLICY.2003.1206966.

[19] Kuzman Katkalov et al. “Model-Driven Development of Information Flow-Secure Systems with IFlow”. In: *International Conference on Social Computing (SocialCom’13)*. IEEE Computer Society, 2013, pp. 51–56. DOI: 10.1109/SocialCom.2013.14.

[20] Jeffrey O. Kephart and David M. Chess. “The Vision of Autonomic Computing”. In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[21] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. “SecureUML: A UML-Based Modeling Language for Model-Driven Security”. In: *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*. Ed. by Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook. Vol. 2460. Lecture Notes in Computer Science. Springer, 2002, pp. 426–441. DOI: 10.1007/3-540-45800-X_33.

[22] Phu Hong Nguyen et al. “An extensive systematic review on the Model-Driven Development of secure systems”. In: *Inf. Softw. Technol.* 68 (2015), pp. 62–81. DOI: 10.1016/j.infsof.2015.08.006.

[23] Ralf H. Reussner et al. *Modeling and Simulating Software Architectures: The Palladio Approach.* The MIT Press, 2016. ISBN: 026203476X.

[24] Kiana Rostami et al. “Architecture-based Assessment and Planning of Change Requests”. In: *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA’15 (part of CompArch 2015).* ACM, 2015, pp. 21–30. DOI: 10.1145/2737182.2737198.

[25] Ahmad-Reza Sadeghi, Christian Wachsmann, and Michael Waidner. “Security and privacy challenges in industrial internet of things”. In: *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015.* ACM, 2015, 54:1–54:6. DOI: 10.1145/2744769.2747942.

[26] Eric Schmieders, Andreas Metzger, and Klaus Pohl. “Runtime Model-Based Privacy Checks of Big Data Cloud Services”. In: *Service-Oriented Computing - 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings.* Ed. by Alistair Barros et al. Vol. 9435. Lecture Notes in Computer Science. Springer, 2015, pp. 71–86. DOI: 10.1007/978-3-662-48616-0_5.

[27] Stephan Seifermann, Robert Heinrich, and Ralf H. Reussner. “Data-Driven Software Architecture for Analyzing Confidentiality”. In: *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019.* IEEE, 2019, pp. 1–10. DOI: 10.1109/ICSA.2019.00009.

[28] Stephan Seifermann et al. “A Unified Model to Detect Information Flow and Access Control Violations in Software Architectures”. In: *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021.* Ed. by Sabrina De Capitani di Vimercati and Pierangela Samarati. SCITEPRESS, 2021, pp. 26–37. DOI: 10.5220/0010515300260037.

[29] Stephan Seifermann et al. “Detecting Violations of Access Control and Information Flow Policies in Data Flow Diagrams”. In: *The Journal of Systems and Software* (2022). accepted, to appear.

[30] Carlos Eduardo da Silva et al. “Self-Adaptive Role-Based Access Control for Business Processes”. In: *12th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017.* IEEE Computer Society, 2017, pp. 193–203. DOI: 10.1109/SEAMS.2017.13.

[31] Gregor Snelting et al. “Checking probabilistic noninterference using JOANA”. In: *it Inf. Technol.* 56.6 (2014), pp. 280–287. DOI: 10.1515/itit-2014-1051.

[32] Khalifa Toumi, César Andrés, and Ana R. Cavalli. “Trust-orBAC: A Trust Access Control Model in Multi-Organization Environments”. In: *Information Systems Security, 8th International Conference, ICISS 2012, Guwahati, India, December 15-19, 2012. Proceedings.* Ed. by Venkat N. Venkatakrishnan and Diganta Goswami. Vol. 7671. Lecture Notes in Computer Science. Springer, 2012, pp. 89–103. DOI: 10.1007/978-3-642-35130-3_7.

[33] Katja Tuma, Riccardo Scandariato, and Musard Balliu. “Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis”. In: *IEEE International Conference on Software Architecture, ICSA 2019, Hamburg, Germany, March 25-29, 2019*. IEEE, 2019, pp. 191–200. DOI: `10.1109/ICSA.2019.00028`.

[34] Dinesh C. Verma et al. “Generative policy model for autonomic management”. In: *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI 2017, San Francisco, CA, USA, August 4-8, 2017*. IEEE, 2017, pp. 1–6. DOI: `10.1109/UIC-ATC.2017.8397410`.

[35] Zeineb Ben Yahya, Farah Barika Ktata, and Khaled Ghédira. “Multi-organizational Access Control Model Based on Mobile Agents for Cloud Computing”. In: *2016 IEEE/WIC/ACM International Conference on Web Intelligence, WI 2016, Omaha, NE, USA, October 13-16, 2016*. IEEE Computer Society, 2016, pp. 656–659. DOI: `10.1109/WI.2016.0116`.